# FUNSOFT Nets: A Petri-Net based Software Process Modeling Language*

Wolfgang Emmerich, Volker Gruhn
Informatik X
Universität Dortmund
Postfach 500 500
D-4600 Dortmund 50
emmerich@udo.informatik.uni-dortmund.de
gruhn@udo.informatik.uni-dortmund.de

February 12, 1996

**Abstract**

We introduce an approach for modeling and analyzing software processes. It is based on describing software processes by FUNSOFT nets. FUNSOFT nets are high level Petri nets which have been particlularly developed to support software process modeling. Their semantics is defined by Predicate/Transition nets. That enables to benefit from standard analysis techniques approved for Predicate/Transition nets. Finally we discuss how the S-invariant analysis method for Predicate/Transition nets helps to reveal properties of software processes described by FUNSOFT nets.

**Keywords**

FUNSOFT nets, semantics definition in terms of Pr/T nets, analysis of FUNSOFT nets, S-invariant analysis method

## 1 Introduction

In this paper we present a result of applying Petri nets to software process modeling. We introduce a kind of high-level Petri nets which is well-suited for describing software process models.

Our basic motivation is the observation that formal languages which enable graphic animation and the use of approved analysis techniques as well as the description of non-determinism and concurrency are sought in various projects aiming at the development of software process modeling languages. While these key requirements lead to Petri nets in general our work revealed that none of the standard high level Petri net types measures up to the more detailed requirements such as tight but still comprehensible representations of software process models. Experiments [Jeg91] have shown that process models defined in terms of Predicate/Transition

*The work described here is partially sponsored by the ESPRIT project ALF and by the EUREKA project ESF

nets [Gen87] (Pr/T nets) tend to become that large that they were neither comprehensible nor manageable even by providing tool-support (e.g. [PSI87]). Thus, we developed a Petri net type, namely **FUNSOFT** nets, which is dedicated to represent software process models. FUNSOFT nets enable to express software process specific properties by single nodes rather than (usually large) subnets in other Petri net types.

The semantics of FUNSOFT nets is defined in terms of Pr/T nets. Thus, the advantage of our approach is to have a high-level application-oriented language and we further benefit from using analysis techniques defined for Pr/T nets to support property proving of software process models.

The next section gives a very concise overview of the relationship between software process modeling and software engineering. Furthermore that section stresses requirements for software process modeling languages. Section 3 introduces FUNSOFT nets. We give an informal explanation of FUNSOFT nets, their definition, their graphic animation, an example, and the semantics definition of FUNSOFT nets in terms of Pr/T nets. Section 4 sketches how the method of S-invariant analysis applied to Pr/T nets can help to reveal properties of software process models described by FUNSOFT nets. Finally, section 5 closes with a conclusion of the work described in this paper.

## 2 Software Process Modeling

Software process modeling is an area of increasing importance [Dow86, Per89]. Its main focus is the description of software process models and the use of such descriptions for communication between people involved in software development, for finding mistakes, for improving the productivity of software development, and at last for increasing the quality of produced software.

The software process is the ordering of activities performed during the creation and evolution of a software system. This explanation of the term *software process* reflects some opinions given in [Dow86].

Regarding this idea of a software process it becomes obvious that there is one software process for each software system that is developed. Several software processes are driven by the same software process model. A software process model describes general features of a class of processes but not those features which are unique for each process (a software process model describes that a software developer has to participate in certain tasks, which software developer actually participates differs from software process to software process). Software process models describe which activities have to be performed by which people, who is allowed to access which documents, and which activities have to be performed when. Beyond the scope of traditional project management descriptions (done by Gantt diagrams or PERT charts [LH89]), software process models serve as the basis for the execution of real software processes and by that enable to retrieve on-line information about the actual state of the process at any time.

Two ways of describing software process models can be distinguished: a model description by means of a formal notation and a narrative or informal description. The wide-spread use of narrative descriptions is emphasized in the following quotation:

> 'Narrative descriptions have been employed by organizations to record their standard

operating procedures - a form of process description' [Kel88].

This statement shows that the idea of describing software processes is not new. Operating procedures have been described for each software development. The problem of most of these descriptions was and is that they lack in precision. They are given in the form of general guidelines and advices like "start with a requirements phase" or "test each module carefully". Therefore, they are the source for a lot of misunderstandings and mistakes. Moreover, it is hardly possible to observe whether an informal software process model is respected during software development itself. Furthermore, no precise analysis techniques can be applied to informal descriptions.

That is why we focus only on formal software process modeling approaches in the following. Subsequently we discuss some of the key requirements for software process modeling languages.

**Representation of non-determinism and concurrency** In software process models several situations exist in which it is unimportant in which way something is done. Instead it is important that it is done in one way or the other. To model this kind of non-determinism is a crucial issue.

Moreover, it is necessary to model that several activities can be executed concurrently. This must be expressible in software process modeling languages. The representation of concurrent activities, for example, can help to find out how many people can be deployed, thus it can be the basis for personnel management.

**Representation of typical entities** A language for the description of software process models must enable the description of essential components of such models, otherwise it does not fulfill its main purpose. Essential components are object types, activity types and some further kind of control conditions. These components are common to all languages for describing software process models. Our investigations have shown that besides object types, activities, and control conditions it is necessary to model predicates of activities [DGS89]. Such predicates are conditions which must be fulfilled before the activity, the predicate is associated with, can be executed. These predicates correspond to the preconditions of activities as defined in [KF87]. Furthermore it must be possible to model the duration of activities.

**Simulation and analysis** Analysis of software process models can contribute to the early detection of errors. By analyzing software process models it is possible to prove specific properties of these models, to detect errors, and to gain deeper insights into the nature of the analyzed software process model.

**Comprehensibility of software process models** A software process modeling language must enable a tight representation of software process models, since software process models are usually quite complex. In order to keep this complexity manageable it is necessary to represent basic entities as single units of description. In the case of software process models this means for example to consider tools as black boxes describing their input/output behavior rather than their internal structure, and the access-order to objects rather than the complete storage mechanisms of an object base. To fulfill the above mentioned requirement these entities must be represented by single units of the language.

The graphic animation of process models can help to understand the structure of the model. Moreover graphic representations of models support the communication between the software process engineer and software developers or customers which participate in the process.

In the next section we introduce FUNSOFT nets. FUNSOFT nets are a software process modeling language that fulfills the discussed requirements.

# 3 FUNSOFT Nets

In this section we introduce a Petri net type which is well-suited for describing process programming fragments. This type of high level Petri nets is called FUNSOFT nets. Some of the concepts of FUNSOFT nets are based on ideas implemented in Function nets [**?**].

## 3.1 Introduction to FUNSOFT Nets

A FUNSOFT net is a tuple $(S, T, F, O, P, J, E, C, A, M_0)$ where $(S, T; F)$ denotes a net [Rei86]. Elements from $S$ are called *channels* and elements from $T$ are called *instances*. $F$ denotes the set of *edges*. Channels are used to model object stores, instances model activities, and describe the relation between activities and object stores.

In order to enable the representation of software process relevant object types and in order to have an extensible set of object types, $O := (O_N \times O_D)$ defines a set of object types. $O_N := TYPID \cup \{BOOL, INTEGER, REAL, STRING\}$ is a set of *type identifiers* and $O_D$ defines a set of *type definitions*. $\{BOOL, INTEGER, REAL, STRING\}$ are predefined types and $TYPID$ denotes identifiers of complex types. For each complex type $O_D$ contains a type definition in the Language $L_{Type}$. In $L_{Type}$ object types are defined analogously to the way object types are defined in the programming language C. For $x \in O_N$, *range(x)* denotes the domain of the type $x$.

Since it is necessary to model that the execution of activities depends on explicit conditions concerning values of tokens which are to be read we introduce activation predicates. Activation predicates can be attached to instances. $P \subseteq (P_N \times P_P)$ denotes a set of activation predicates. $P$ is called the *activation predicate library*. Each predicate from the library consists of a name from the set $P_N$ and a list of parameters from $P_P$. A predicate can be used to ensure that an instance can only fire if tokens with certain properties are available in the preset of the instance.

In order to get less complex nets (provided that a major part of the complexity is given by the number of nodes) executable code can be attached to transitions in many net classes. This code is executed whenever the transition is fired (e.g. ML in Colored Petri Nets). In these approaches it is assumed that by firing a transition a number of fixed tokens is consumed from each place in the preset and another number of fixed tokens is produced for each place in the postset. This assumption has to be weakened since activities occurring in software processes do not have usually this firing behavior.
In our approach instances can be inscribed with *jobs* and if an instance occurs, the corresponding job is executed. Jobs are members of the *job library* $J$, $J := (J_N \times J_{FI} \times J_{FO} \times J_P)$. Jobs have got names from $J_N$. The input firing behavior $j_{fi} \in \{ALL, MULT, COMPLEX\}$ and the output firing behavior $j_{fo} \in \{ALL, SOME, DET, MULT, COMPLEX\}$ of a job describe how a transition to which that job is attached behaves. The formal semantics definition of jobs will be given in terms of Pr/T Nets in subsection 3.5.4. An input firing behavior $ALL$ for example indicates that the job reads one token from each channel of the preset of the instance it is assigned to. An output firing behavior $MULT$ for example indicates that the

job writes a natural number $n$ to the first channel of the postset of the instance to which the job is assigned and that it writes $n$ tokens to the second channel of the postset of the instance the job is assigned to. An example of a job with a *COMPLEX* output firing behavior is the *compile*-job. The result of a compilation is either a successfully compiled module or a module that has to be re-edited together with a failure report.
The job parameterization consists of input parameters and output parameters.

Edges are inscribed by two functions $E := (E_T, E_N)$. We distinguish information flow and control flow as well as reading tokens by removing and by copying and with that again enhance the modeling power of our language. $E_T$ assigns an *edge type* from the set $\{IN, CO, OU, ST, FI\}$ to each edge. An edge $e \in F$ with $E_T(e) \in \{IN, CO, OU\}$ models information flow and an edge $e$ with $E_T(e) \in \{ST, FI\}$ models control flow. If $E_T((s,t)) = IN$ and $t$ occurs, a token is removed from $s$. If $E_T((s,t)) = CO$ and $t$ occurs, a token is copied from $s$. The function $E_N$ defines an *edge numbering*. The edge numbering is needed for checking consistency between the parameterization of the attached job and the object types assigned to the channels in the pre- and postset.

$C := (C_A, C_T)$ defines two functions which assign attributes to channels. $C_A$ attaches an *access attribute* to each channel. The possible values are $\{FIFO, LIFO, RANDOM\}$. Thereby we are able to model different kinds of access by a simple channel attribute. The access attribute defines the order in which tokens are removed from the channel. $FIFO$ denotes a 'First-In-First-Out' order, $LIFO$ defines a 'Last-In-First-Out' order. Channels $s$ with $C_T(s) = RANDOM$ behave like places in P/T-Nets do.
$C_T$ attaches an *object type* from $O$ to each channel. Each channel $s$ can only be marked with tokens of type $C_T(s)$.

An instance can be annotated with up to four inscriptions. They are assigned by four functions $A := (A_J, A_P, A_T, A_W)$. $A_J$ assigns a job to each agency. $A_J(t)$ denotes the job assigned to the instance $t$. $A_P$ is a partial function which assigns predicates from the predicate library $P$ to instances. $A_P(t)$ denotes the predicate assigned to the instance $t$. The function $A_T$ assigns a positive real value or the value 0 to instances. $A_T(t)$ denotes the time consumption of instance $t$, it quantifies the amount of time which passes between reading tokens from the preset of $t$ and writing tokens to the postset of $t$. The function $A_W$ assigns one of the values $\{PIPE, NOPIPE\}$ to instances. The *pipelining attribute* $A_W(t)$ defines whether the instance $t$ models a pipeline or not. If $A_W(t) = PIPE$, $t$ can fire without having finished previous firings. Otherwise $t$ must finish each firing before it can occur again.

The initial marking of the net is defined by the function $M_0$, which assigns a set of tuples from $(range(C_T(s)) \times I\!N)$ to each channel $s$. The first component of each tuple is the object value, the second one is a natural number. This natural number defines an order of the tokens. This order is required for channels $s$ with $C_T(s) \in \{LIFO, FIFO\}$, since the access to tokens marking such channels depends on their arrival order.

## 3.2 Syntax of FUNSOFT Nets

**Definition 3.1 Parameterizations of activation predicates and jobs**
*The languages $L_{AP}$ and $L_{JP}$ which define the parameterizations of activation predicates and jobs are defined by the following grammar. The language $L_{AP}$ is generated with the start symbol $< ActivationPredicateParameter >$ and for $L_{JP}$ the start symbol is $< JobParameter >$.*

$$
\begin{array}{lcl}
< Char > & ::= & a|\ldots|z|A|\ldots|Z \\
< ComplexTypeid > & ::= & < Char > \{< Char >\}_0^* \\
< SimpleTypeid > & ::= & BOOL|INTEGER|REAL|STRING \\
< Typeid > & ::= & < SimpleTypeid > | < ComplexTypeid > \\
< Parameter > & ::= & < Typeid > \{\times < Typeid >\}_0^* \\
< ActivationPredicateParameter > & ::= & < Parameter > \\
< JobParameter > & ::= & < Parameter > \rightarrow < Parameter >
\end{array}
$$

A word of the language $L_{AP}$ is for example $person \times REAL$, a word of the language $L_{JP}$ is for example $person \times REAL \rightarrow BOOL$.

**Definition 3.2 FUNSOFT nets**
*Let $L_{Type}$ be the language for defining token types and let $TYPID$ denote the set of correct type identifiers. Let $L_{AP}$ and $L_{JP}$ be languages for defining parameterizations of activation predicates and jobs as defined above. A tuple $FS=(S, T, F, O, P, J, A, C, E, M_0)$ is a FUNSOFT net, iff:*

1. $(S, T; F)$ *is a net*

2. $O \subseteq$ $(O_N \times O_D)$ *defines object types by*
   $O_N =$ $\{BOOL, INTEGER, REAL, STRING\} \uplus TYPID$
   *is a set of type identifiers*
   $O_D \subseteq$ $L_{Type}$ *is a set of type definitions for* $O_N$

3. $P \subseteq$ $(P_N \times P_P)$ *defines a library of predicates where*
   $P_N$ *is a set of predicate names*
   $P_P \subseteq L_{AP}$ *is a set of predicate parameterizations*

4. $J \subseteq$ $(J_N \times J_{FI} \times J_{FO} \times J_P)$ *is a library of jobs with*
   $J_N$ *is a set of job names*
   $J_{FI} =$ $\{ALL, MULT, COMPLEX\}$
   $J_{FO} =$ $\{ALL, SOME, DET, MULT, COMPLEX\}$
   *describe the input and the output firing behavior*
   $J_P \subseteq$ $L_{JP}$ *ist a set of parameterizations*

5. $E =$ $(E_T, E_N)$ *defines edge annotations with*
   $E_T :$ $\begin{cases} F \cap (S \times T) \rightarrow \{IN, CO, ST\} \\ F \cap (T \times S) \rightarrow \{OU, FI\} \end{cases}$ *function assigning an edge type*
   $E_N :$ $F \rightarrow I\!N$ *defines an order on the pre- and postset of each instance by*
   $\forall_{(t,s),(s',t') \in F}\ E_N(t,s) \leq |t\bullet| \wedge E_N(s',t') \leq |\bullet t'|$
   $\forall_{(s,t),(s',t) \in F}\ E_N(s,t) \neq E_N(s',t)$
   $\forall_{(t,s),(t,s') \in F}\ E_N(t,s) \neq E_N(t,s')$

6. $C =$ $(C_A, C_T)$ *defines channel annotations by*
   $C_A :$ $S \rightarrow \{RANDOM, LIFO, FIFO\}$
   *function assigning access attributes*
   $C_T :$ $S \rightarrow O_N$ *function assigning object types*

7. $A =$ $(A_J, A_P, A_T, A_W)$ *defines instance annotations by*
   $A_J :$ $T \to J$ *function assigning jobs*
   $A_P :$ $T \to P$ *partial function assigning activation predicates*
   $A_T :$ $T \to \mathbb{R}_0^+$ *function assigning time consumptions*
   $A_W :$ $T \to \{PIPE, NOPIPE\}$ *function assigning pipelining attributes*

8. $M_0$ *defines the initial marking by*
   $M_0 : S \to \mathcal{P}((\bigcup_{o \in O_N} range(o)) \times \mathbb{N})$
   *iff* $M_0$ *respects the channel types defined by* $C_T$.

### 3.3 Graphical Representation of FUNSOFT Nets

The net structure of a FUNSOFT net is graphically represented as usual: channels are drawn as circles, instances as rectangles and edges as arrows.

The firing behaviors of jobs provide some information about the behavior of jobs during their execution. Firing behaviors are displayed as follows:

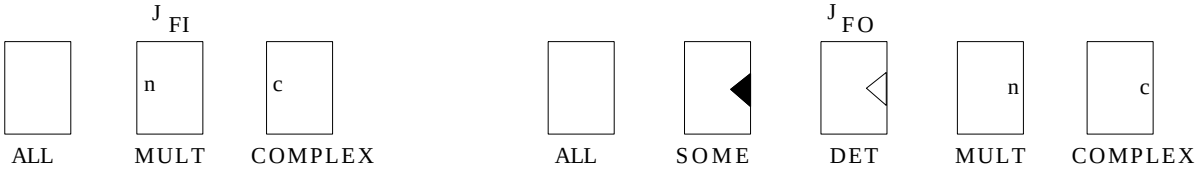
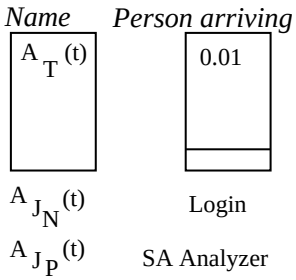
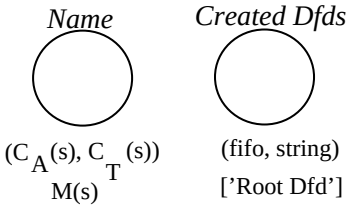



Figure 1: Graphical representation of the firing behavior of jobs

Since each job has an input and an output firing behavior graphical symbols for both kinds of behaviors can be combined.




The figure left to this paragraph shows where attributes of an instance depicted as a rectangle are displayed. The pipelining attribute $PIPE$ is indicated by a horizontal line in the lower part of the rectangle. If this line is missing, the instance has the attribute $NOPIPE$. If $A_T(t) = 0$ it is omitted. On the right hand of the figure an example is given with $A_J(t) = Login$, $A_P(t) = SA\ Analyzer$, $A_W(t) = PIPE$ and $A_T(t) = 0.01$. The instance name is optional




The graphical representation of channel attributes and of initial markings is shown in the figure left to this paragraph. The example on the right of that figure shows a channel $s$ with $C_T(s) = STRING$, $C_A(s) = FIFO$ and $M(s) = \{('RootDfd', 1)\}$. The channel name is optional.

The type of an edge is drawn near to the arrow representing the edge. If it is omitted an edge $(s, t)$ is of the type $IN$ and an edge $(t, s)$ is of the type $OU$. Edge numbers are written under edges. For the remainder of this document edge numbers are omitted whenever the parameter position of tokens read from or written to channels is clear.

### 3.4 A FUNSOFT Net Example

This subsection introduces an example showing how a requirements analysis phase of a software process can be modeled by means of FUNSOFT nets.

The graphical representation of this example is enabled by using two kinds of hierarchies, which were introduced in [HJS89], namely instance substitution and channel fusion.

In the following nets, instances inscribed with *DEC* denote that an instance is refined by a subnet. In the subnet each channel which is connected with the refined instance of the supernet is also depicted. This helps to recognize the interrelation of the two nets. Such channels are called ports of the subnet. For example the instance *Requirements Analysis* in Figure 3 is refined by the net in Figure 4 and the channels *start* and *sa ready* are the ports of that net.

Channels represented by a dotted circle are fusioned to a channel of the same name, which is drawn by a solid circle and which appears in the same net or in another subnet. For example the channels *working sa-analyzers* of Figure 4 are fusioned to the channel *working sa-analyzer* of the net shown in Figure 2.

One of the object types used in this software process model is for example the object type `person` which is a record containing a name, a salary, the number of hours worked at the current day, number of hours worked in total, and a role. All object types used in this example are explained in Table 2.

For an informal description of the jobs used in this example confer to Table 1.

Figure 2 shows an example of a model of project management activities.

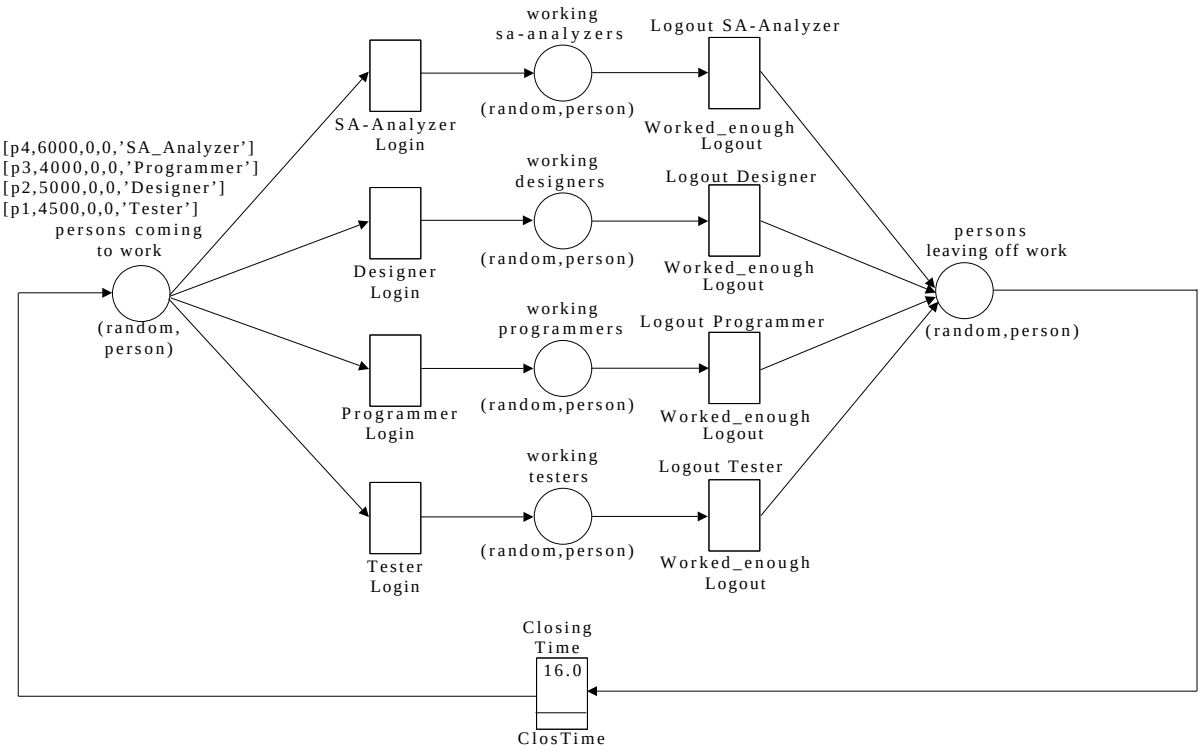



Figure 2: Personnel Management

| Jobname | *Firing behavior* | (*Input parameter*)→(*Output parameter*) |
|---|---|---|
| Informal description of the job | | |
| Login | $(all, all)$ | $(person) \rightarrow (person)$ |
| Reads from the input channel a token representing a person, initializes the number *person.worked_today* to zero and fires the token into the output channel. | | |
| Logout | $(all, all)$ | $(person) \rightarrow (person)$ |
| Reads from the input channel a token representing a person, adds the number *person.worked_today* to *person.worked_total* and fires the result into the outpt channel. | | |
| ClosTime | $(all, all)$ | $(person) \rightarrow (person)$ |
| Reads a token from the input channel and fires it to the output channel. | | |
| CreEmpSA | $(all, all)$ | $(bool) \rightarrow (samodel)$ |
| Reads a control token and fires an object of the type samodel which is initialized with empty lists into the postset. | | |
| EditSA | $(all, all)$ | $(person \times samodel) \rightarrow (person \times samodel)$ |
| Reads a token from the type samodel and a token representing a person from the preset, increases *person.worked_today* by the time consumption of the instance and fires the tokens to the postset. | | |
| AnalyzeSA | $(all, some)$ | $(samodel) \rightarrow (samodel \times samodel \times string)$ |
| Reads a token from the preset and fires it randomly either to the first or to the second output channel. If the job fires to the second output channel a string modeling an error report is fired to the third output channel. | | |
| Less | $(all, all)$ | $(person \times string) \rightarrow (person)$ |
| Reads a tokens from the preset, increases *person.worked_today* by the time consumption of the instance and fires the modified token to the postset. | | |
| Decide | $(all, some)$ | $(person \times samodel) \rightarrow (person \times samodel \times samodel)$ |
| Reads a token from the first channel of the preset, and fires the increased component *person.worked_today* into the first channel of the postset. Moreover, a token is read from the second channel of the preset. This token is fired randomly into the second or into the third channel of the postset. | | |
| Move | $(all, all)$ | $(samodel) \rightarrow (samodel \times samodel)$ |
| Reads a token from the preset, duplicates it and fires one to each output channel. | | |

Table 1: Jobs used in the example

Tokens in this net represent persons and are of the object type `person`. Persons can be in the states *coming to work*, *working* and *leaving work*. The job *Login* reads a token of the type person from the preset, initializes the number `worked_today` and writes it to the postset. All instances having the job *Login* are inscribed with different activation predicates which are explained in Table 3. They check the role of a person and guarantee, that the channels in the postset are only marked with persons having the checked role.

The job *Logout* reads tokens of the type `person`, adds the number `worked_today` to the number `worked_total` and writes the new token into the preset. The activation predicate *worked_enough* is attached to all instances inscribed with the job *Logout*. The predicate checks whether `worked_today` is greater than 8.0 (hours). The job of the instance *Closing Time* has got a time consumption of 16.0 hours and the instance allows pipelining, so that persons are removed from *persons leaving off work* and written to *persons coming to work* with a delay of 16.0 hours.

Figure 3 shows a net which models a waterfall driven software process.

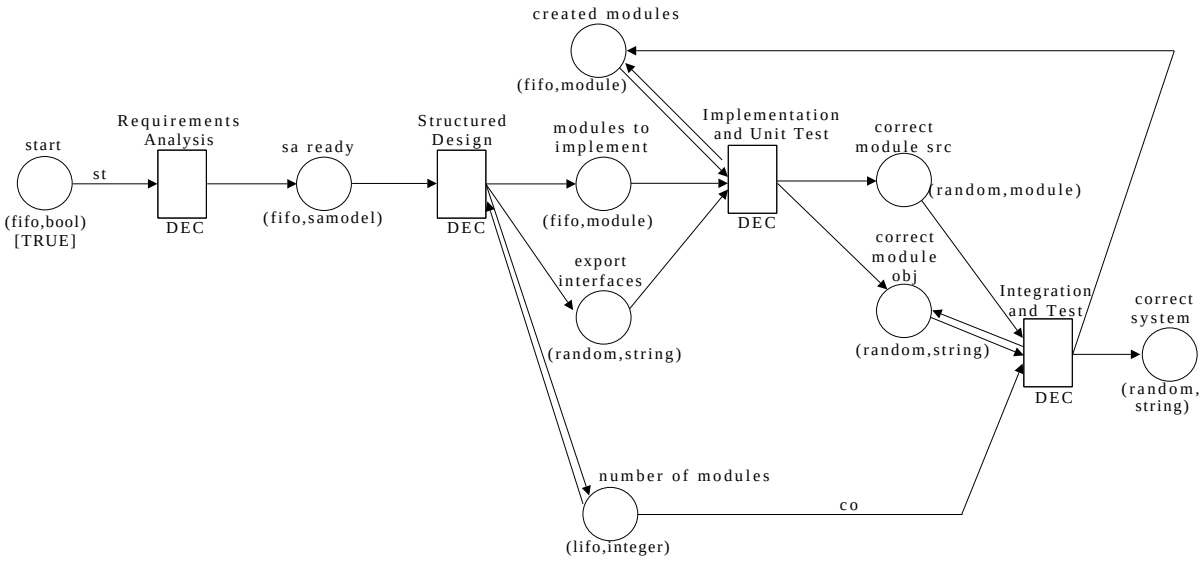



Figure 3: A waterfall driven software process modeled with FUNSOFT nets

Every instance of this net is refined by a subnet. Its channels define the ports between the subnets. In the following we explain the refinement of the instance *Requirements Analysis* which is shown in Figure 4.

This net models a requirements analysis phase following the Structured Analysis method [dM79]. The ports to the instance refined by this net, namely *start* and *sa ready* are showed on the left respective right sight of this Figure.

If the instance *Create empty SA-Model* occurs it reads the control token from its preset and fires a token of the object type *samodel* into its postset. The instance *Edit SA* models the editing of the SA model by sa-analyzers. The time to carry out this work is modeled by the time consumption of 4.0 hours.

The check whether a SA model holds the consistency criteria proposed by de Marco for SA documents is modeled by the instance *Analyze SA*. As this activity can be performed in the background, this instance has the pipelining attribute value *PIPE*. The instance *Analyze SA*

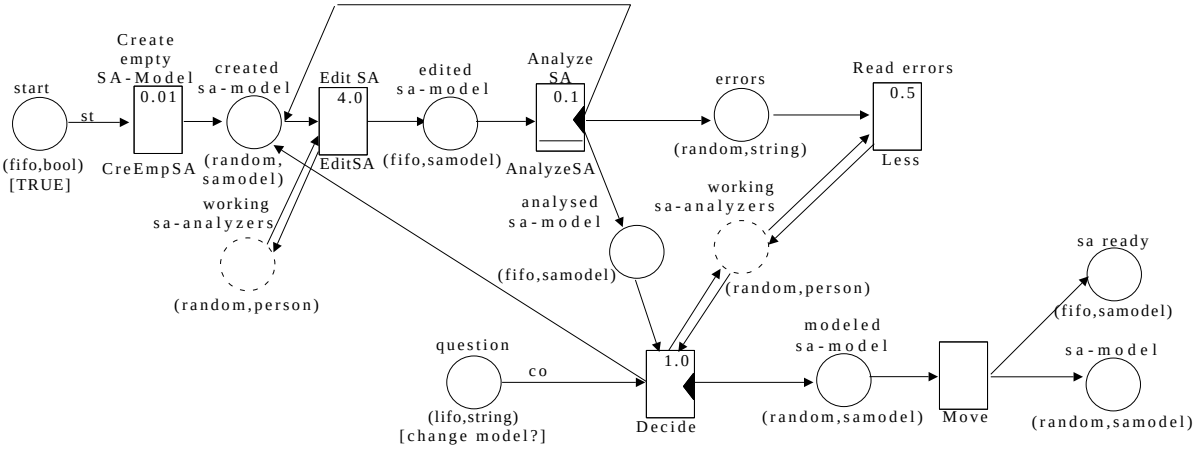



Figure 4: Requirements Analysis with SA

produces either an error message or an analyzed SA model, which is considered to be correct (For detailed description cf. Table 1).

Reading of error messages by sa-analyzers is modeled by the instance *Read errors*. The fact that the last decision on the correctness should be held by sa-analyzers, is modeled by the instance holding the job *Decide*. This instance fires randomly the token from *analyzed sa-model* either to *created sa-model* or to *modeled sa-model*. The instance holding the job *Move* models the duplication of SA models in order to enable persons in later phases to read this document.

### 3.5 Unfolding FUNSOFT Nets to Predicate/Transition Nets

In the beginning of section 3 we gave a short and informal explanation of the semantics of FUNSOFT nets. This informal explanation does not enable us to define dynamic properties like *activation, firing behavior* or the *reachability set*. A formal semantics definition of FUNSOFT nets is a prerequisite for analyzing them by standard Petri net analysis techniques.

In this section we describe an algorithm which is a 'local simulation' [Sta87] of FUNSOFT nets by Pr/T nets with 'Many-sorted Structures', 'Multi-Sets', and the 'Weak Transition Rule' as proposed by Genrich [Gen87].

Pr/T nets resulting from applying the algorithm to FUNSOFT nets and the FUNSOFT nets themselves are related by a net morphism [SR87]. The construction of Pr/T nets out of FUNSOFT nets is called *unfolding* in the following. The result of applying this unfolding to a net element is called the *unfolded net element*, the result of applying it to a FUNSOFT net is called the *unfolded FUNSOFT net*.

In this section we describe the unfolding of FUNSOFT net components and how the unfolded components are assembled. Furthermore we use the result of this construction to define the dynamic behavior of FUNSOFT nets. That builds the foundation for calling FUNSOFT nets a type of Petri nets.

In the following we give a rough sketch how the unfolding of FUNSOFT net elements is performed. In the beginning a first-order structure building the support of the Pr/T net is provided. Secondly, the object types are mapped onto variable predicates. They are used

in the unfolding of channels. Thirdly, for each access attribute value a Pr/T net is defined. Fourthly, jobs are described by Pr/T nets. These Pr/T nets define the input and the output firing behaviors of a job formally. Moreover, each activation predicate is translated into a static predicate. They are assigned to transitions in unfolded instances. The unfolding of instances is mainly determined by the Pr/T net defining the job attached to these instances. The Pr/T nets resulting from unfolding channels and instances are connected according to the edge type of the edge connecting channel and instance in the FUNSOFT net. At last, formal sums of tuples of constants derived from the marking of the FUNSOFT net, are attached to places of the unfolded channels. This way of unfolding FUNSOFT nets to Pr/T nets is described in detail in the rest of this section.

### 3.5.1 Support

The signature of the supporting structure used in the Pr/T nets encompasses the sorts: Bool, Int, Real, Str, Set(Type), List(Type) as well as a set of commonly used functions and predicates for these sorts.

### 3.5.2 Object types

The primitive type identifiers $BOOL, INTEGER, REAL, STRING$ are translated into the unary variable predicates $\langle Bool \rangle, \langle Int \rangle, \langle Real \rangle, \langle Str \rangle$. Type identifiers denoting complex object types are translated into variable predicates reflecting the structure of the object types. The construction of records is implemented by building tuples over the variable predicates. The construction of list is mapped onto the abstract type List. Correspondingly the construction of sets is mapped onto the abstract type Set.

In Table 2 the object types and the corresponding variable predicates used in the example given in the previous subsection can be found.

| Names ($O_N$) | Definitions ($O_D$) | Variable predicates |
|---|---|---|
| person | `struct person {char *name; float salary; float worked_today; float worked_total; char *role;}` | $\langle Str, Real, Real, Real, Str \rangle$ |
| dfdlist | `struct dfdlist {char *dfd; struct dfdlist *next;}` | $\langle List(Str) \rangle$ |
| dalist | `struct dalist {char *da; struct dalist *next;}` | $\langle List(Str) \rangle$ |
| msplist | `struct msplist {char *msp; struct msplist *next;}` | $\langle List(Str) \rangle$ |
| samodel | `struct samodel {dfdlist *dfds; dalist *das; msplist *msps;}` | $\langle List(Str), List(Str), List(Str) \rangle$ |

Table 2: Type definitions for complex object types

### 3.5.3 Channels

As mentioned above channels can have different access attribute values, namely $FIFO$, $LIFO$ and $RANDOM$. The unfolding of channels with different access attribute values results in Pr/T nets with different internal structures. The elements of the surface of unfolded channels are places. These places are called *ports*. We distinguish between *input ports* (places from which transitions of the Pr/T net read tokens) and *output ports* (places to which transitions of the Pr/T net write tokens). Independent from the access attribute value unfolded channels always have the same ports. Thus there is only one way to connect unfolded channels to unfolded instances. That fact allows us to restrict ourselves to describe the unfolding of a $FIFO$ channel in this paper.

Figure 5 shows the generic Pr/T net which results from unfolding a FUNSOFT channel $s$ with $C_A(s) = FIFO$.

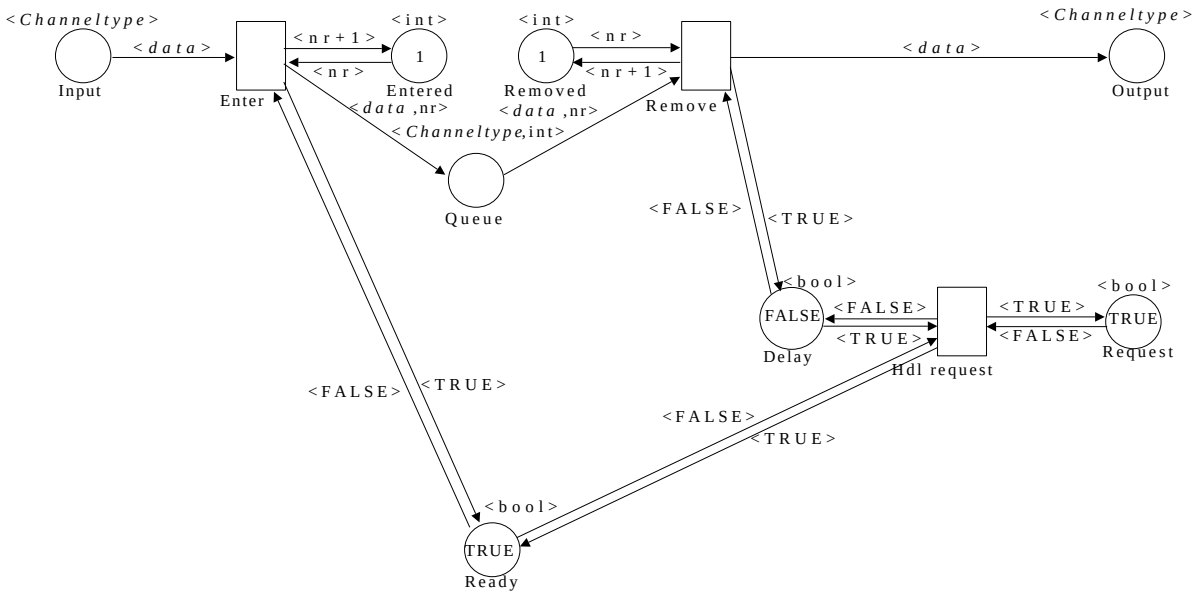



Figure 5: Generic Unfolding of a Channel $s$ with $C_A(s) = FIFO$

The variable predicates $\langle Channeltype \rangle$ (used for the places Input and Output) are replaced by the predicates derived from the object type $C_T(s)$. Accordingly the inscriptions $\langle data \rangle$ are replaced by the symbolic sums representing objects of the type $C_T(s)$. How the variable predicates are derived from $C_T(s)$ has been described in the previous subsection. The inscription of the edges are derived analogously. By replacing the variable predicate $\langle Channeltype \rangle$ and the $\langle data \rangle$ inscriptions we obtain a concrete Pr/T net.

The places *Input*, *Output*, *Request* and *Ready* are the ports of the Pr/T net. Exactly these places are connected with other transitions when a Pr/T net for a whole FUNSOFT net is assembled.

Tokens are written to *Input* by unfolded instances of the preset of $s$. Tokens are read from *Output* by unfolded instances of the postset of $s$.

If the port *Output* is marked, it is marked with exactly that token which resides on the FUNSOFT channel for the longest time. The places *Entered* and *Removed* are marked with exactly one natural number, initially they are marked with 1. The number marking the place

*Entered* shows how often tokens were fired into the original FUNSOFT channel $s$. The number marking the place *Removed* shows how many tokens were already removed from $s$.

By marking the place *Input* the transition *Enter* is enabled. This transition reads a number from *Entered* and a token from *Input*, builds a tuple of both and fires this tuple to the place *Queue*. In the same firing the number of the place *Entered* is increased and the boolean value residing on *Ready* is set to $TRUE$. This shows that one token is ready to be processed by transitions outside the unfolded channel and that token to transitions outside of this net and that *Input* can be marked again. In so far *Ready* can be considered as a semaphore for *Input*. This enables the definition of an order of tokens arriving on *Input*.

The transition *Remove* removes tokens from *Queue* exactly in their arrival order. *Remove* is enabled, if the second component of the tuple marking *Queue* equals the number marking the place *Removed* and if *Output* is unmarked. Whether *Output* is marked or not is indicated by the marking of the place *Delay*. If *Output* is unmarked *Delay* is marked with $FALSE$. Firing *Remove* means to cut the second component of the token read from *Queue* and to write the first component to *Output*, and to increase the number marking the place *Removed*, and to set *Delay* to $TRUE$.

If an arbitrary transition reads a token from *Output* it additionally sets the markings of *Request* and *Ready* to $FALSE$. Thereby the transition *Hdl request* is enabled. *Hdl request* sets the marking of *Request* and *Ready* to $TRUE$ again, the marking of *Delay* to $FALSE$. Afterwards the next token can be fired to *Output*.

The Pr/T net of Figure 5 guarantees the required *FIFO* access to tokens of the channel $s$.

### 3.5.4 Jobs

Jobs are attached to instances. At last the structure of the job $A_J(t)$ determines the structure of the Pr/T net resulting from unfolding $t$. The jobs are defined in terms of Pr/T nets, thus for each job of the job library a Pr/T net must be provided. This Pr/T net defines the semantics of the job. Figure 6 shows as an example the Pr/T net for the Job *CheckDfd*. The input firing behavior of *CheckDfd* is $ALL$, its output firing behavior is $SOME$ and the parameterization is $(string \rightarrow string \times string \times string)$. *CheckDfd* reads a *string* which models a data flow diagram from its input channel and writes it non-deterministic either to the first or to the third output channel. In the latter case an error message is written to the second output channel.

Corresponding to the parameterization we can find a place inscribed with the variable predicate $\langle Str \rangle$ on the left side. Moreover, we find there two places with the names *OReady1* and *ORequest1*. These places represent the input ports of the job *CheckDfd*. These places are merged with ports of unfolded channels of the preset of instances to which the job *CheckDfd* is assigned. This merging takes place when a complete Pr/T net is assembled. On the right we find three places inscribed with the variable predicates $\langle Str \rangle$. Together with the places inscribed with *Ready1, Ready2, Ready3* they build the output ports of the job *CheckDfd*. In between input and output ports we find

- a transition which reads tokens from the places on the left and which checks the activation predicate.
- a subnet which manipulates the input tokens and creates the output tokens.

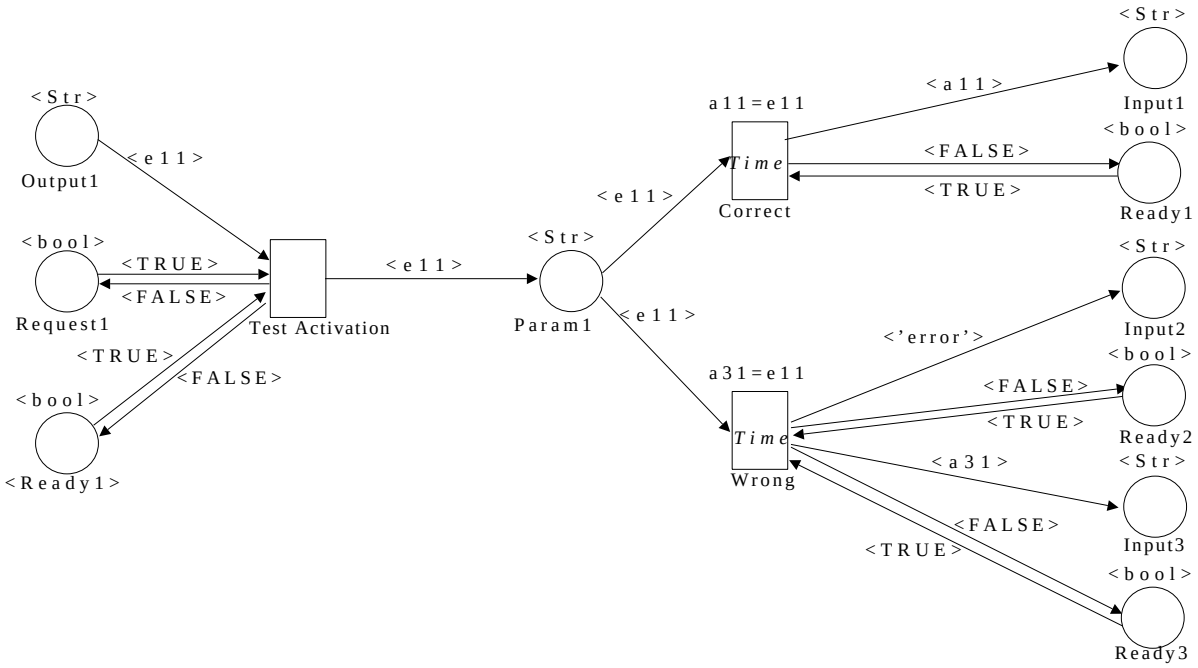



Figure 6: Pr/T net defining the job CheckDfd

For each job exactly one transition is labelled with *Test Activation* and at least one transition is labelled with *Time*. Static predicates and time consumptions are assigned to these labelled transitions during the unfolding of instances.

### 3.5.5 Activation predicates

Each used activation predicate is translated into a first-order formula. These formulae are used as static predicates of transitions *Test Activation* in unfolded instances. In the following Table all activation predicates of the example mentioned above are given and translated into first-order formulae.

| Activation predicate | | Formula | Description |
|---|---|---|---|
| Name | Parameter | | |
| SA_Analyzer | person | 'SA-Analyzer' = e15 | TRUE, if person has role 'SA-Analyzer' |
| Designer | person | 'Designer' = e15 | TRUE, if person has role 'Designer' |
| Programmer | person | 'Programmer' = e15 | TRUE, if person has role 'Programmer' |
| Tester | person | 'Tester' = e15 | TRUE, if person has role 'Tester' |
| worked_enough | person | e13 > 8.0 | TRUE, if person has worked today at least 8 units |

Table 3: Transformation of activation predicates into formulae

### 3.5.6 Instances

For explaining the unfolding of instances we refer to the mentioned Pr/T net representations of jobs. The unfolding of an instance $t$ is essentially determined by the Pr/T net representing the job $A_J(t)$. What has to be supplemented are components reflecting the specific instance

attributes, namely the formula derived from the activation predicate, a time consumption, and a place for defining the pipelining behavior.

The formula derived from the activation predicate is attached as static predicate to that transition of the Pr/T net defining the attached job which is labelled with *Test Activation*, the time consumptions of instances is assigned as time consumption to the transitions labelled with *Time*. The semantics of time consumptions of transitions are the same as defined in [Ram74]. Due to the internal structure of all nets defining jobs no conflicts are resolved by activation times. Thus the application of analysis techniques for non timed Petri nets is not affected by this definition of time consumptions.

Figure 7 shows a simplified net resulting from unfolding an instance and the extensions caused by the pipelining attribute having the value $PIPE$. The dashed box contains the extensions.

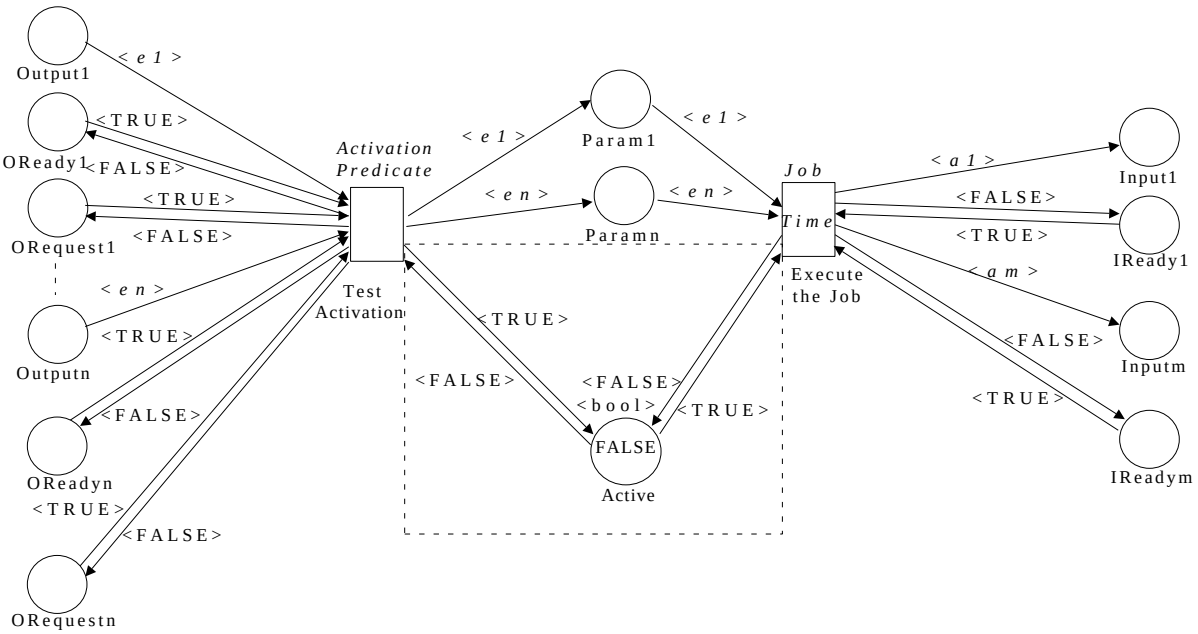



Figure 7: Extensions of a net defining a instance $t$ with $A_W(t) = PIPE$

The additional place inscribed with *Active* is connected to the transitions labelled with *Test Activation* and *Execute the Job*. It guarantees, that the transition *Test Activation* is only enabled if the transition *Execute the Job* has finished the firing. For a pipelining attribute value $NOPIPE$ the additional place and its adjacent edges are omitted.

### 3.5.7 Edges

If an edge $e$ connects a channel and an instance corresponding edges have to connect the unfolded channel and the unfolded instance. To connect an unfolded channel and an unfolded instance means to merge their ports. For edges $(s,t)$ we have to merge the output ports of the unfolded channel $s$ with the input ports of the unfolded instance $t$. For edges $(t,s)$ the input ports of the unfolded channel $s$ are merged with the output ports of the unfolded instance $t$. Edges $e$ with $E_T(e) = ST$ are treated as edges with the edge type $IN$, edges $e$ with $E_T(e) = ST$ are treated as edges with the edge type $OU$. In the following Figure 8 it is depicted how edges with edge type $IN$ and $OU$ are represented in the Pr/T net.

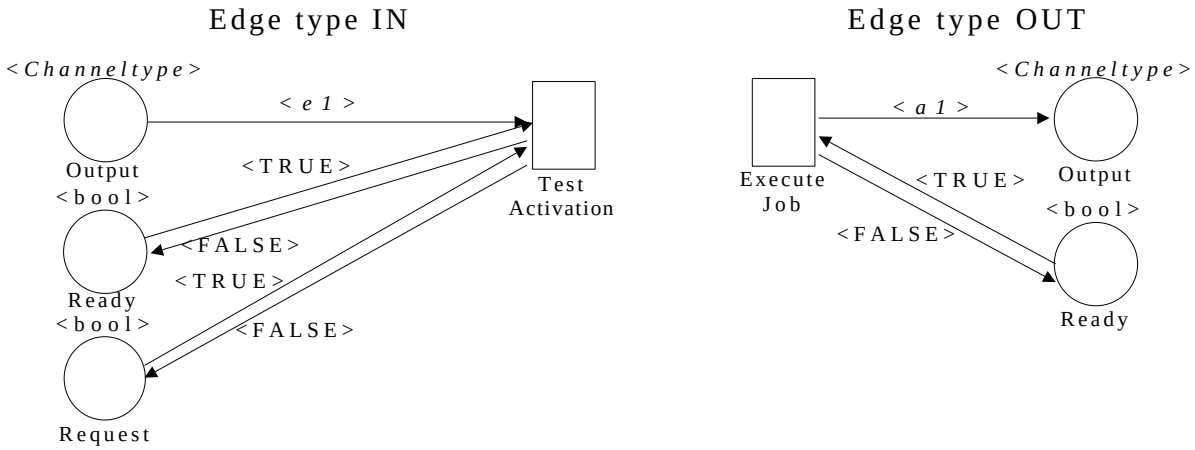



Figure 8: Pr/T net representation of edges with types IN and OU

Edges $(s,t)$ with $E_T(s,t) = CO$ are represented as edges from the type $IN$, but the edges in the Pr/T net connecting *Request* and *Test Activation* and vice versa are omitted and the edge connecting *Test Activation* and *Ready* is inscribed with $\langle TRUE \rangle$.

### 3.5.8 Initial Marking

Transforming the marking of a channel $s$ means to mark several places of the unfolded channel $s$. In the following we describe how the initial marking $M(s)$ of a channel $s$ with $C_A(s) = FIFO$ is transformed into the initial marking of the unfolded channel. The transformation of initial markings of channels $s$ with $C_A(s) \in \{LIFO, RANDOM\}$ is done analogously.

If the marking $M(s)$ is empty the initial marking of the unfolded channel corresponds to the one given in Figure 5. Otherwise the places of the unfolded channels are marked as follows: The place *Input* is unmarked. The places *Delay*, *Request* and *Ready* are initially marked with $TRUE$. The place *Entered* is marked with $|M(s)| + 1$ and the place *Removed* is marked with 2. The place *Output* is marked with that token, that has to be accessed at first, i.e. with that token whose second component equals 1. The place *Queue* is marked with all tokens of $M(s)$ which are not accessed at first.

Figure 9 shows as an example the transformation of the marking $M(s) = \{(4,1),(3,2),(7,3)\}$ of a channel $s$ with $C_T(s) = INTEGER$ and $C_A(s) = FIFO$.

### 3.5.9 Assembling unfolded channels and instances

The following Figure 10 shows an Pr/T nets resulting from assembling an unfolded instance $t$ with $A_J(t) = CheckDfd$ and its input and output channels. The input channel as well as the output channels have the access attribute value $RANDOM$.

## 3.6 Dynamic behavior of FUNSOFT nets

Let in the following $f$ denote the unfolding of channels and instances. $f(s)$ denotes an unfolded channel $s$ and $f(t)$ denotes an unfolded instance $t$. For a FUNSOFT net $FS$, $f(FS)$ denotes

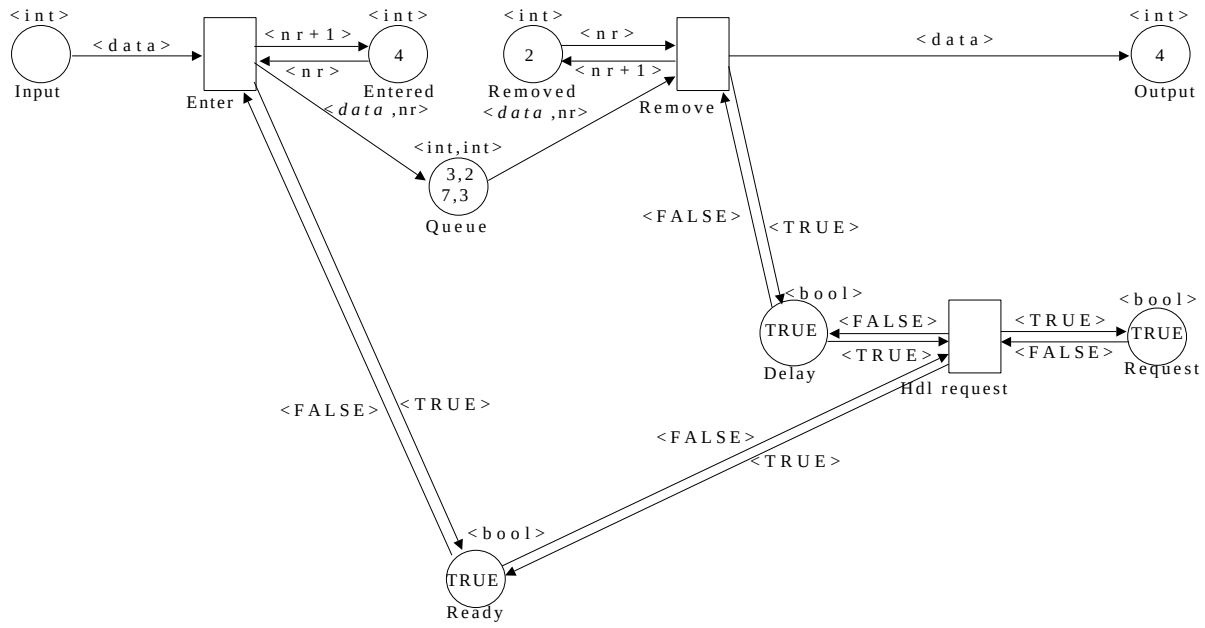



Figure 9: Translation of a Marking for $C_A = FIFO$

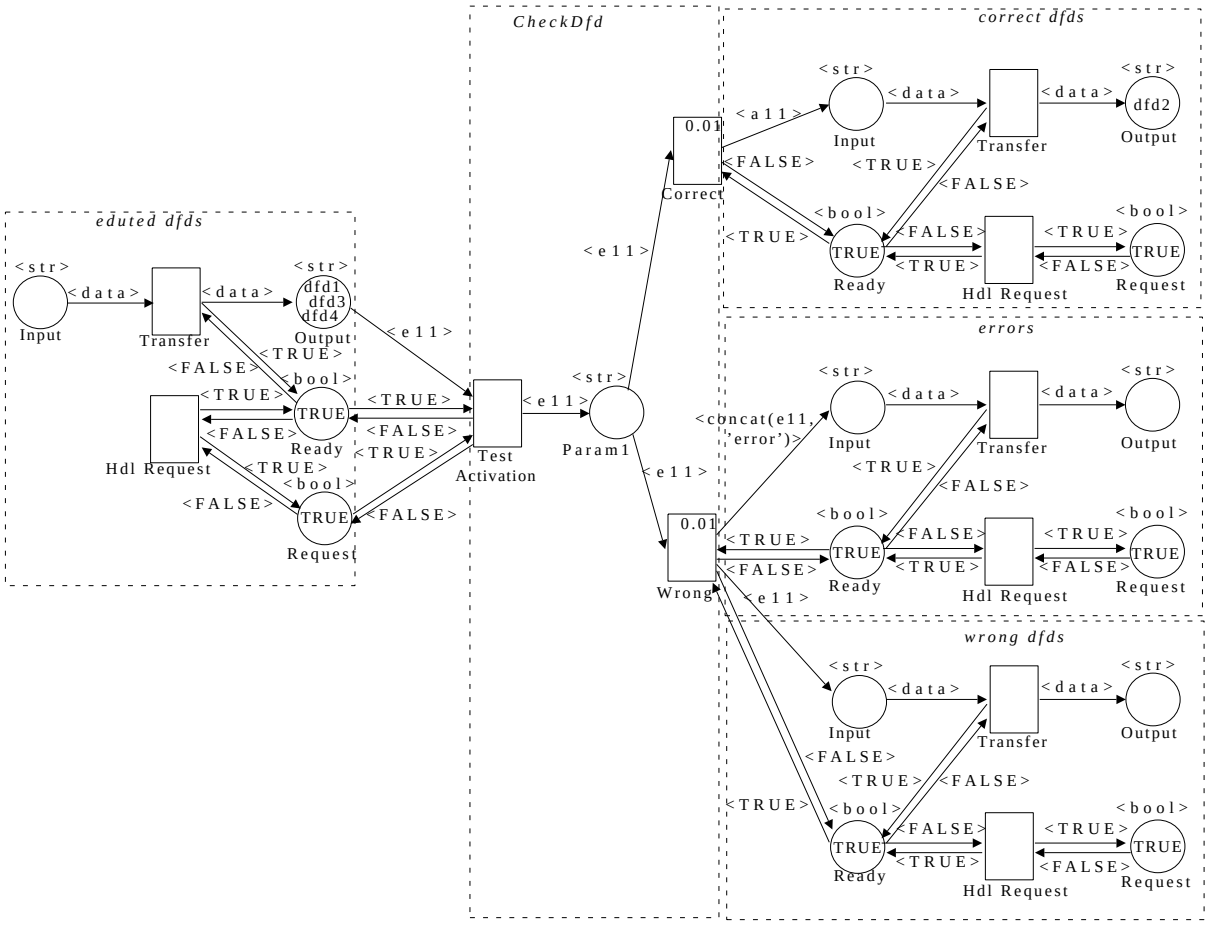



Figure 10: An assembled Pr/T net

the unfolding of $FS$ (including the translation of the initial marking). If a FUNSOFT net $FS$ is marked with a marking $M$ that is different from the initial marking of $FS$, then $g_{FS}(M)$ denotes the unfolding of $FS$ under the marking $M$.

**Definition 3.3 Marking of FUNSOFT nets**
*Let $FS = (S, T, F, O, P, J, A, C, E, M_0)$ denote a FUNSOFT net. The annotation*

$$M : S \rightarrow \mathcal{P}((\bigcup_{o \in O} range(o)) \times I\!N)$$

*is called Marking of FS, if it respects $C_T$.*

**Definition 3.4 Enabled instances**
*Let $FS = (S, T, F, O, P, J, A, C, E, M_0)$ denote a FUNSOFT net. An instance $t$ is enabled under a marking $M$, iff a transition in $f(t)$ is enabled in $f(FS)$.*

**Definition 3.5 Firing rule**
*Let $FS = (S, T, F, O, P, J, A, C, E, M_0)$ denote a FUNSOFT net. If $t$ is enabled under a marking $M$, the result of its occurrence $M'$ is determined by the firing of $t_1, \ldots, t_n \in f(\bullet t) \cup f(t) \cup f(t\bullet)$ for which holds:*

$$g_{FS}(M)[_{t_1,\ldots,t_n}\rangle g_{FS}(M')$$

*with $g_{FS}(M')$ enabling no transition of $f(t) \cup f(t\bullet)$. If $M'$ results by firing instance $t$ from $M$, we write $M[t\rangle M'$.*

**Definition 3.6 Reachability set**
*Let $FS = (S, T, F, O, P, J, A, C, E, M_0)$ denote a FUNSOFT net and $M$ be a marking of $FS$. The reachability set $M[\rangle$ of $M$ is the smallest set for which holds:*

1. $M \in M[\rangle$
2. $\forall_{t \in T} : M[t\rangle M' \Rightarrow M' \in M[\rangle$

# 4 Verification of FUNSOFT net properties

In this section we explain how FUNSOFT nets representing software process models can be analyzed.

Before we explain the applied analysis techniques in detail we sketch the method for obtaining software process relevant results. This method is depicted in Figure 11.

The diagram of Figure 11 shows that our approach towards analysis of software process models is driven by the software process model specific relevance of expected results. That means we start with defining software process model properties which we are interested in from a software process modeling point of view. These properties are transformed into corresponding properties of FUNSOFT nets. In some cases these properties can be verified by applying algorithms directly to FUNSOFT nets, in other cases it is necessary to unfold FUNSOFT nets to Pr/T nets and to verify the corresponding properties of the unfolded net.

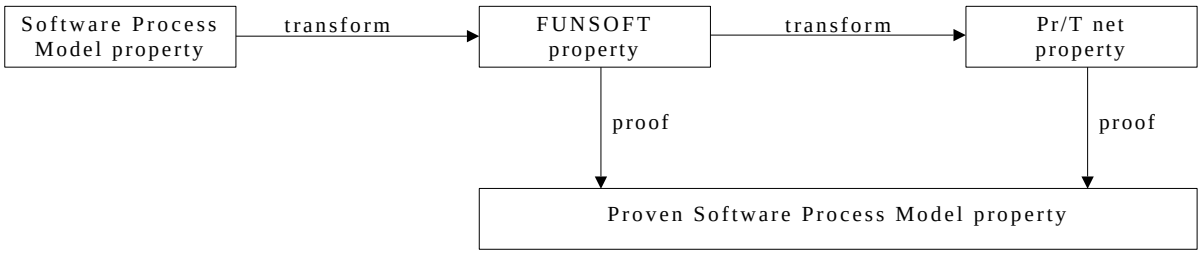



Figure 11: Verification method for FUNSOFT nets

In principle we distinguish between the verification of dynamic properties of software process models and the verification of static properties of software process models. While the verification of dynamic properties is based on reachability trees built for FUNSOFT nets [DG91], the verification of static properties is based on the examination of the net topology of FUNSOFT nets and on the S-invariant analysis method for Pr/T nets [Gen87].

Within this paper we restrict ourselves to the examination and interpretation of S-invariants of Pr/T nets that result from unfolding FUNSOFT nets.

Reverting to the net depicted in Figure 2 one interesting software process model property is whether one of the persons participating in software development may disappear somewhere in the process (which means that someone does not participate in the software process, what obviously reveals an error in the software process model).

The corresponding FUNSOFT net property is strict conservativeness, i.e. the number of tokens in the net shown in Figure 2 remains invariant.

This means that the analysis relevant predicates of the unfolded FUNSOFT net are covered by an S-invariant.

**Definition 4.1 analysis relevant predicates of $f(N)$:**
*Let $N$ be a FUNSOFT net. The set $AR(f(N))$ of* **analysis relevant predicates of** $f(N)$ *is defined as follows:*

$$AR(f(N)) = \bigcup_{s \in S} \sigma(s) \;\; with \;\; \sigma(s) = \begin{cases} s\_input, s\_output, & if\, C_A(s) = RANDOM \\ s\_input, s\_queue, s\_output, & if\, C_A(s) = FIFO \\ s\_input, s\_stack, s\_output, & if\, C_A(s) = LIFO \end{cases}$$

The previous definition identifies exactly those predicate as analysis relevant which can be marked with representations of FUNSOFT tokens. In contrast to these there are further predicates which are used for synchronization purposes (cf. Figure 5).

Table 4 shows the matrix $\underline{C}$ representing the Pr/T net resulting from unfolding the FUNSOFT net shown in Figure 2 and the solutions of $\underline{C}^T \cdot \underline{i} = \underline{0}$. The solutions have been figured out by means of the tool described in [KL84] which is based on [Mev81].

The solution $i_3$ is an S-invariant since it is variable free. As it covers all analysis relevant predicates (i.e. the predicates *pctw inp, pctw out, wsa inp, wsa out, wde inp, wde out, wpr inp, wpr out, wte inp, wte out, plw inp, plw out*) it proves the strict conservativeness of the FUNSOFT net and thereby the requested property of the software process model. Correspondingly, we can be sure that the number of people remains invariant in the net shown in Figure 2.

| | | pctw | | LiSA | | LiDe | | LiPr | | LiTe | | wsa | | | wde | | | wpr | | | wte | | | LoSA | | LoDe | | LoPr | | LoTe | | plw | | CT | | Solutions | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | tran | hdrq | act | exj | act | exj | act | exj | act | exj | tran | hdrq | | tran | hdrq | | tran | hdrq | | tran | hdrq | | act | exj | act | exj | act | exj | act | exj | tran | hdrq | act | exj | $t_1$ | $t_2$ | $t_3$ |
| pctw | inp | -per | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | per | | | |
| | out | per | | -san | | -des | | -pro | | -tes | | | | | | | | | | | | | | | | | | | | | | | | | f-t | | | |
| | rdy | -f+t | -f+t | f-t | | f-t | | f-t | | f-t | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | req | | -f+t | f-t | | f-t | | f-t | | f-t | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| LiSA | param1 | | | san | -san | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| wsa | inp | | | | san | | | | | | | -san | | | | | | | | | | | | | | | | | | | | | | | | | |
| | out | | | | | | | | | | | san | | | | | | | | | | | | -per | | | | | | | | | | | | | |
| | rdy | | | | f-t | | | | | | | -f+t | -f+t | | | | | | | | | | | f-t | | | | | | | | | | | | | |
| | req | | | | | | | | | | | | -f+t | | | | | | | | | | | f-t | | | | | | | | | | | | | |
| LiDe | param1 | | | | | des | -des | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| wde | inp | | | | | | des | | | | | | | | -des | | | | | | | | | | | | | | | | | | | | | | |
| | out | | | | | | | | | | | | | | des | | | | | | | | | | | -per | | | | | | | | | | | |
| | rdy | | | | | | f-t | | | | | | | | -f+t | -f+t | | | | | | | | | | f-t | | | | | | | | | | | |
| | req | | | | | | | | | | | | | | | -f+t | | | | | | | | | | f-t | | | | | | | | | | | |
| LiPr | param1 | | | | | | | pro | -pro | | | | | | | | | -pro | | | | | | | | | | | | | | | | | | | |
| wpr | inp | | | | | | | | pro | | | | | | | | | pro | | | | | | | | | | | | | | | | | | | |
| | out | | | | | | | | | | | | | | | | | -f+t | -f+t | | | | | | | -per | | | | | | | | | | | |
| | rdy | | | | | | | f-t | | | | | | | | | | | -f+t | | | | | | | f-t | | | | | | | | | | | |
| | req | | | | | | | | | | | | | | | | | | -f+t | | | | | | | f-t | | | | | | | | | | | |
| LiTe | param1 | | | | | | | | | tes | -tes | | | | | | | | | -tes | | | | | | | | | | | | | | | | | |
| wte | inp | | | | | | | | | | tes | | | | | | | | | tes | | | | | | | | | | | | | | | | | |
| | out | | | | | | | | | | | | | | | | | | | -f+t | -f+t | | | | | | | | | | | | | | | | |
| | rdy | | | | | | | | | f-t | | | | | | | | | | | -f+t | | | | | | | | | | | | | | | | | |
| | req | | | | | | | | | | | | | | | | | | | | | | per | -per | | | | | | | | | | | | | |
| LoSA | param1 | | | | | | | | | | | | | | | | | | | | | | | | per | -per | | | | | | | | | | |
| LoDe | param1 | | | | | | | | | | | | | | | | | | | | | | | | | | per | -per | | | | | | | | |
| LoPr | param1 | | | | | | | | | | | | | | | | | | | | | | | | | | | | per | -per | | | | | | |
| LoTe | param1 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | per | -per | | | | |
| plw | inp | | | | | | | | | | | | | | | | | | | | | | | | per | | per | | per | | per | | | | | |
| | out | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -per | | | |
| | rdy | | | | | | | | | | | | | | | | | | | | | | | | f-t | | f-t | | f-t | | -f+t | | f-t | | -per | |
| | req | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -f+t | | f-t | | per | |
| CT | param1 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | per | -per | | | 1 |

$f$ $\langle FALSE \rangle$

$t$ $\langle TRUE \rangle$

$per$ $\langle Str, Real, Real, Real, Str \rangle$

$san$ $\langle Str, Real, Real, Real, 'SA\_Analyzer' \rangle$

$des$ $\langle Str, Real, Real, Real, 'Designer' \rangle$

$pro$ $\langle Str, Real, Real, Real, 'Programmer' \rangle$

$tes$ $\langle Str, Real, Real, Real, 'Tester' \rangle$

Table 4: Incidence matrix of Pr/T net resulting from unfolding the FUNSOFT net from Fig. 2

This short sketch of how we exploit standard Petri net techniques has shown that our approach is a very pragmatic one. Our research of looking at software process model properties which can be proven by standard Petri net techniques is an ongoing activity, since we do not believe that all possibilities of analysis techniques have been exploited yet.

# 5 Conclusions and further work

We described a result of combining knowledge about software process modeling and high-level Petri nets, namely FUNSOFT nets. They enable the exploitation of standard Petri net analysis techniques. In this way results concerning the application area can be obtained on a sound mathematical basis.

The practical use of FUNSOFT nets is supported by an environment called MELMAC [DG90, DG91]. MELMAC provides the necessary tool support for graphical development and animation, for simulation, the calculation of quantity restricted reachability trees, and the S-invariant Pr/T net analysis, which is based on the tool PetSI [KL84].

Our current efforts concentrate on the extension of MELMAC by further analysis tools. Moreover, our approach to software process modeling, analysis, and execution is currently used in the Esprit project ALF and the Eureka project ESF.

**Acknowledgements:**

We want to thank the project group MELMAC for implementing essential parts of the above mentioned environment, W. Deiters for contributing basic ideas to it, Prof. W. Schäfer and Dr. B. Holtkamp for giving helpful comments on earlier versions of this paper, and H. Jeibmann, J. Cramer, S. Wolf, K.-J. Vagts, and B. Peuschel for intensive discussions about process modeling. The authors acknowledge the contribution to this paper from all the members of the ALF consortium, who are: GIE Emeraude (France), CSC (Belgium), Computer Technologies Co., CTC (Greece), Grupo de Mecanica del Vuelo, S.A. (Spain), International Computers Limited (United Kingdom), University of Nancy-CRIN (France), University of Dortmund-Informatik X (Germany), Cerilor (France), Universite de Catholique de Louvain (Belgium) and University of Dijon-CRID (France).

# References

[DG90] W. Deiters and V. Gruhn. *Managing Software Processes in MELMAC.* In *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments*, pages 193–205, Irvine, California, USA, December 1990.

[DG91] W. Deiters and V. Gruhn. *Software Process Model Analysis Based on FUNSOFT Nets. Mathematical Modeling and Simulation*, (8), May 1991.

[DGS89] W. Deiters, V. Gruhn, and W. Schäfer. *Systematic Development of Generic Formal Software Process Models.* In *Proceedings of the* $2^{nd}$ *European Software Engineering Conference, Coventry, UK*, Berlin, FRG, September 1989. Springer. Appeared as Lecture Notes in Computer Science 387.

[dM79] T. de Marco. *Structured Analysis and System Specification.* Yourdon Press, 1979.

[Dow86] M. Dowson, editor. *Iteration in the Software Process - Proceedings of the $3^{rd}$ International Software Process Workshop*, Beckenridge, Colorado, USA, November 1986.

[Gen87] H.J. Genrich. *Predicate/Transition Nets.* In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Central Models and Their Properties*, pages 208–247, Berlin, FRG, 1987. Springer. Appeared in Lecture Notes on Computer Science 254.

[HJS89] P. Huber, K. Jensen, and R.M. Shapiro. *Hierarchies in Coloured Petri Nets.* In *Proc. of the $10^{th}$ Int. Conf. on Application and Theory of Petri Nets*, pages 192–209, Bonn, FRG, 1989.

[Jeg91] R. Jegelka. *Evaluierung der Software-Prozeßmodellierungs-Sprache FUNSOFT-Netze und der Software-Prozeßmanagement-Umgebung MELMAC (in German).* 1991. Diplomarbeit, University of Dortmund.

[Kel88] M.I. Kellner. *Representation Formalisms for Software Process Modelling.* In *Proceedings of the $4^{th}$ International Software Process Workshop*, Moretonhampstead, Devon, UK, May 1988.

[KF87] G.E. Kaiser and P.H. Feiler. *An Architecture for Intelligent Assistance in Software Development.* In *Proceedings of the $9^{th}$ International Conference on Software Engineering*, Monterey, California, 1987.

[KL84] R. Kujansuu and M. Lindquist. *Efficient Algorithms for computing S-Invariants for Predicate/Transition Nets.* In *Proceedings of the $5^{th}$ International Conference on Application and Theory of Petri Nets*, 1984.

[LH89] L. Liu and E. Horowitz. *A Formal Model for Software Project Management. IEEE Transactions on Software Engineering*, 15(10), October 1989.

[Mev81] H. Mevissen. *Algebraische Bestimmung von S-Invarianten in Prädikat/Transitions-Netzen.* Working report of the GMD no 81.01, Gesellschaft für Mathematik und Datenverarbeitung, Bonn, FRG, 1981.

[Per89] D.E. Perry, editor. *Proceedings of the $5^{th}$ International Software Process Workshop*, Kennebunkport, Maine, USA, September 1989.

[PSI87] PSI GmbH, Berlin, FRG. *NED Release 2.0*, 1987.

[Ram74] C. Ramchandani. *Analysis of asynchronous concurrent systems by timed Petri Nets.* PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, 1974.

[Rei86] W. Reisig. *Petrinetze.* Springer, Berlin, FRG, 1986.

[SR87] E. Smith and W. Reisig. *The Semantics of a Net is a Net.* In *Concurrency and Nets*, Berlin, FRG, 1987. Springer.

[Sta87] P. Starke. *On the mutual simulatability of different types of Petri nets.* In *Concurrency and Nets*, pages 481–495, Berlin, FRG, 1987. Springer.