Software Process Modelling with FUNSOFT Nets

Wolfgang Emmerich, Volker Gruhn
Computer Science
Software Technology
University of Dortmund
P.O. Box 500 500
D-4600 Dortmund 50
Federal Republic of Germany

October 12, 1995

Abstract

In this paper we introduce FUNSOFT nets. FUNSOFT nets are high level Petri nets which are well-suited for software process modelling. We define the semantics of FUNSOFT nets in terms of Pr/T nets. Thus we enable the use of standard Petri net analysis techniques for examining software process model properties. We point out which analysis techniques are of interest from a software process modelling point of view. Moreover, we give an example for a software process model represented by a FUNSOFT net and we point out which tools for editing and analysing FUNSOFT nets are available.

Keywords

FUNSOFT nets, semantics definition in terms of Pr/T nets, analysis of FUNSOFT nets, application of FUNSOFT nets, FUNSOFT net tools

Topics

Higher-level net models, application of nets to software process modeling, analysis of nets

Contents

1 Introduction

In this paper we describe a result of combining knowledge in the area of software process modelling and in the area of Petri net research[1]. 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 modelling languages. While these key requirements lead to Petri nets in general our work partially carried out in the frame of the ESPRIT project ALF [BBCD89] and the EUREKA project ESF [SW88] revealed that none of the standard Petri net types measures up to the more detailed requirements like integration of existing tools into software process models, description of complex object types, and distinction between data and control flow. Therefore, we developed a Petri net type, namely **FUNSOFT** nets, which is well-suited for representing software process models.

The semantics of FUNSOFT nets is defined in terms of Pr/T nets. Thus we benefit from using FUNSOFT nets as application-oriented language and from using analysis techniques defined for Pr/T nets for proving properties of software process models.

The next section focuses on giving a very concise overview about the relation of software process modelling to software engineering in general, about some software process modelling approaches of prime interest and it critically judges the results obtained in software process modelling up to now. Section 3 introduces FUNSOFT nets. This introduction contains 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 analysis techniques which are well-known for standard Petri nets can be applied to FUNSOFT nets. Moreover, section 4 shows how results obtained by standard analysis techniques can be interpreted to reveal properties of the software process model represented by a FUNSOFT net. This section does not give a complete list of properties which are worthwhile to be proven, but it puts emphasis on how to obtain software process relevant results. Section 5 gives a short overview about some tools which enable the use of FUNSOFT nets. Finally, section 6 concludes our work.

2 Software Process Modelling

Software process modelling is an area of increasing interest [3ISPW, 4ISPW, 5ISPW]. Its main focus is to describe software process models and to use such descriptions for communication between people involved in software development, for finding mistakes, and at last for improving the productivity of software development and increasing the quality of produced software.

The software process is the sequence of activities performed during the creation and evolution of a software system.

This explanation of the term *software process* reflects some opinions given in [3ISPW] and

[1]The work described here is partially sponsored by the ESPRIT project ALF and by the EUREKA project ESF

[4ISPW].

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. Well-known examples of software process models are the waterfall model [Royc70], the spiral model [Boeh88], and prototyping models [BCG83]. 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. Software process models build the basis for the software process itself. Software processes can be considered as software developments following a particular software process model.

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' [Kell88].

This statement shows that the idea of describing software processes is not a new one. Of course, operating procedures have been described for each software development. The problem of the most of these descriptions was and is that they are lacking in preciseness. 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 if an informal software process model is respected during software development itself and no precise analysis techniques can be applied to informal descriptions.

That is why we focus only on formal software process modelling approaches in the following.

One of the first ideas about how to model software processes was described by Osterweil. His paper *Software Processes Are Software Too* [Oste87] raised a lot of controversial discussions. The approach introduced by Osterweil is the **Process Programming** approach. The main idea is to think of and model software processes as software.

That is one of the main motivations of Lehman to criticize the Process Programming approach. He points out that it will hardly be possible to describe the creative process of software development a priori [Lehm87], that means before starting the software development itself.

Several approaches to software process modelling can be found [Dows87, KF87, TBCO88, Robe88, HJPS89, DGS89]. They differ mainly in the used language for describing software process models and in the tools provided for modelling software processes. Even though the most of these approaches claim not to follow the idea of Process Programming it must be stated that they focus mainly on those parts of software process models which can be supported by existing tools and which are well-understood (a typical example for such a part is an *edit-compile-test* cycle). Other parts which belong to software development as well as these understood parts, such as *discussing a system's design with a customer* or *doing a design review* are not considered.

For the rest of this paper we explicitly point out that we restrict ourselves to the well-understood parts of software process models (or to be precise to their modelling and to their analysis).

Another research area in software process modelling deals with different phases of building and using software process models [MGDS90]. Whenever such phases are distinguished the analysis of software process models is identified as an important activity in building and using software process models [Kell89, Fink89, MGDS90].

Since we introduce a software process modelling language in the next section we discuss some of the key requirements for languages which are to be used for describing software process models in the following.

Graphic animation A software process model is very complex. Therefore a graphic animation can help to understand the structure of the model. The need for user-friendly representations is emphasized in [Kell88, Dows86a].

Representation of concurrency and non-determinism In software process models several situations exist in which it is of no importance in which way something is done, but only that it is done in one of several ways. It is necessary to model this kind of non-determinism.

Moreover, it is necessary to model that several activities can be executed concurrently. This must be expressible. The representation of activities which can be carried out in parallel can help to find out how many people can be deployed, thus it can be the basis for personnel management. The need to model this concurrency is emphasized in [BB88, Tayl86].

Simulation and analysis Analysis of software process models can contribute to the early detection of errors. By analysing 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. The need for employing analysis techniques in the examining software process models is stressed in [Kell89, Fink89].

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, since these components are used in all languages for describing software process models. Our experiments have shown that besides object types, activities, and control conditions it is necessary to model predicates of activities. Such predicates are conditions which must be fulfilled before the activity which is associated with the predicate can be executed. These predicates correspond to the preconditions of activities as defined in [KF87].

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 [Godb83]. Before developing FUNSOFT nets existing high level Petri net types were examined. In principle the standard high level Petri net types such as Pr/T nets [Genr86] and Coloured Petri nets [Jens86] enable the representation of software process models. But due to the fact that these Petri net types were not developed for software process modelling they do not measure up to the detailed requirements for software process modelling languages. The modelling of already existing tools causes problems as well as the definition of time consumptions and the definition of different ways of accessing S-elements. That is why FUNSOFT nets were developed. The essential advantage of FUNSOFT nets is that they enable a dense representation of software process models. Thus the representation of software process models appears less complex since some complexity is hidden in inscriptions of net elements.

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 [Reis86]. Elements from S are called *channels* and elements from T are called *instances*.

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. $O_N := TYPID \dot{\cup} \{BOOL, INTEGER, REAL, STRING\}$ 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 [KR77]. 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.

In many net classes executable code can be attached to transitions (e.g. ML in Coloured Petri Nets) in order to get less complex nets (provided that complexity is measured in number of nodes). In our approach instances can be inscribed with *jobs* and if an instance occurs, the corresponding job is executed. A job can be considered as an atomic and well-understood activity. 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. For each job an input firing behaviour $j_{fi} \in \{ALL, MULT, COMPLEX\}$ and an output firing behaviour $j_{fo} \in \{ALL, SOME, DET, MULT, COMPLEX\}$ describe informally how the job behaves when it is executed. An input firing behaviour ALL for example indicates that the job reads tokens from all channels of the preset of the instance it is assigned to. An output firing behaviour $MULT$ 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. The job parameterization J_P defines the types of tokens, which are read and written by the job. The job parameterization consists of input parameters and output parameters which are separated by an horizontal arrow.

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. 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 $\{FIFO, LIFO, RANDOM\}$. 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 $(range(C_T(s)) \times \mathbb{N})$ 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 $< JobParameter >$.

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

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*

$$
\begin{array}{lll}
O_N = & \{BOOL, INTEGER, REAL, STRING\} \cup TYPID \\
& \textit{is a set of type identifiers} \\
O_D \subseteq & L_{Type} \textit{ is a set of type definitions for } O_N
\end{array}
$$

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 behaviour
$J_P \subseteq$ L_{JP} *ist a set of parameterizations*

5. $E =$ (E_T, E_N) *define 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*
$A_J :$ $T \rightarrow J$ *function assigning jobs*
$A_P :$ $T \rightarrow P$ *partial function assigning activation predicates*
$A_T :$ $T \rightarrow I\!R_0^+$ *function assigning time consumptions*
$A_W :$ $T \rightarrow \{PIPE, NOPIPE\}$ *function assigning pipelining attributes*

8. M_0 *defines the initial marking by*
$M_0 : S \rightarrow \mathcal{P}((\bigcup_{o \in O_N} range(o)) \times I\!N)$
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 behaviours of jobs provide some information about the behaviour of jobs during their execution. Firing behaviours are displayed as follows:

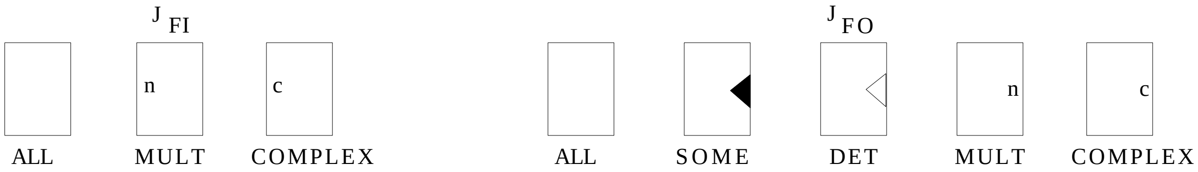
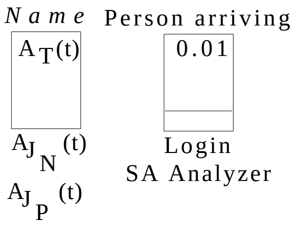
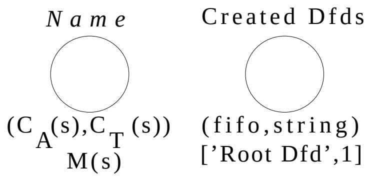


Figure 1: Graphical representation of the firing behaviour of jobs

Since each job has an input and an output firing behaviour graphical symbols for both kinds of behaviours 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 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 modelled by means of FUNSOFT nets. The given nets are part of an example provided in [Emme89], where a complete waterfall driven software process is modelled 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. For example the instance *Requirements Analysis* in Figure 3 is refined by the net in Figure 4. 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.

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. They check the role of a person and guarantee, that the channels in the postset are only marked with persons having the checked role.

Jobname	Firing behaviour	(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 modelling 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

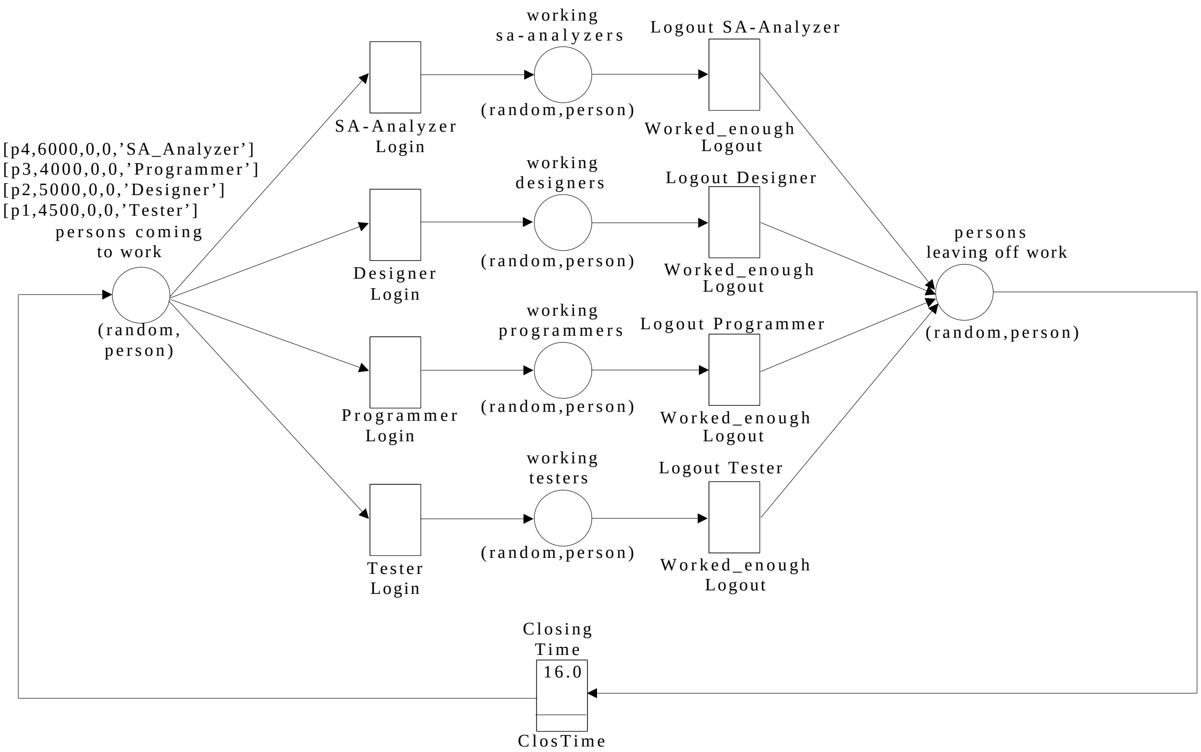


Figure 2: Personnel Management

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.

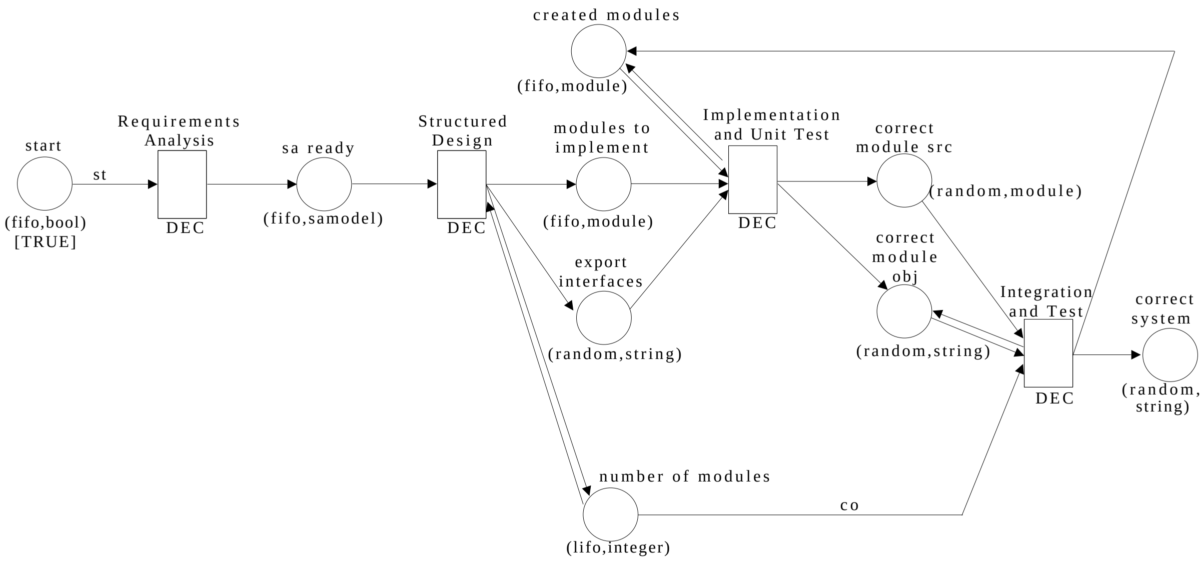


Figure 3: A waterfall driven software process modelled with FUNSOFT nets

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.

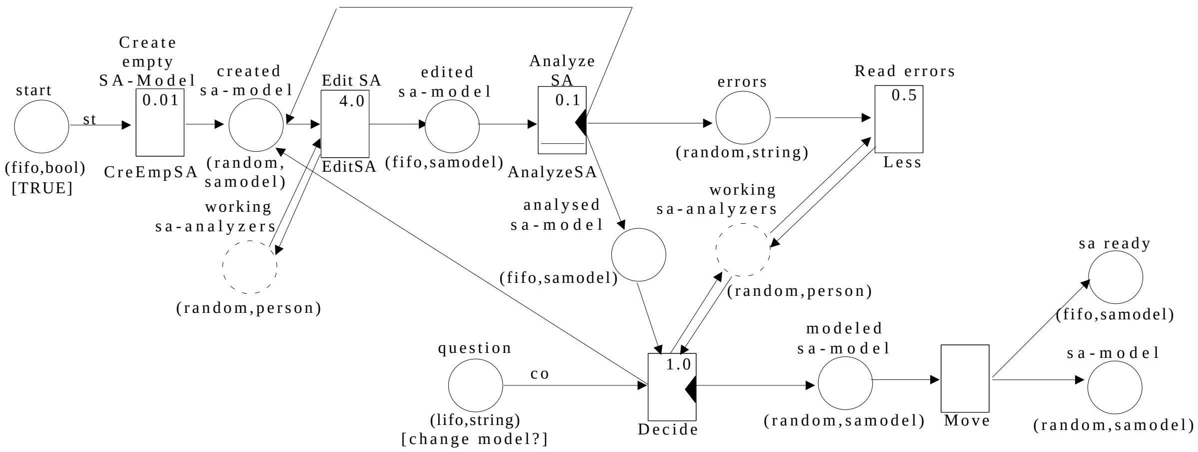


Figure 4: Requirements Analysis with SA

This net models a requirements analysis phase following the Structured Analysis method [DeMa79]. 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 modelled 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 modelled 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* 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 modelled by the instance *Read errors*. The fact that the last decision on the correctness should be held by sa-analyzers, is modelled 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 behaviour* or the *reachability set*. A formal semantics definition of FUNSOFT nets is a prerequisite for analysing them by standard Petri net analysis techniques.

In this section we describe an algorithm which is a 'local simulation' [Star87] of FUNSOFT nets by Pr/T nets with 'Many-sorted Structures', 'Multi-Sets', and the 'Weak Transition Rule' as proposed by Genrich [Genr86].

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 behaviour of FUNSOFT nets. That builds at last 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 behaviours 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 $C_A(s) = FIFO$.

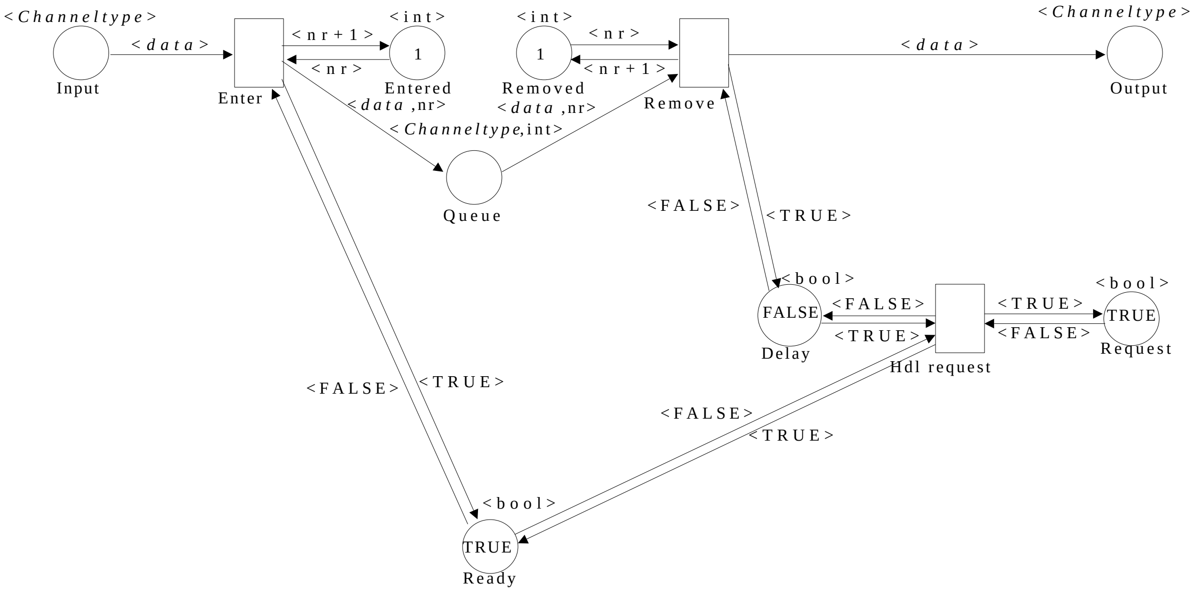


Figure 5: Generic Unfolding of a Channel s with $C_A(s) = FIFO$

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 semaphor 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 behaviour of *CheckDfd* is ALL, its output firing behaviour 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.

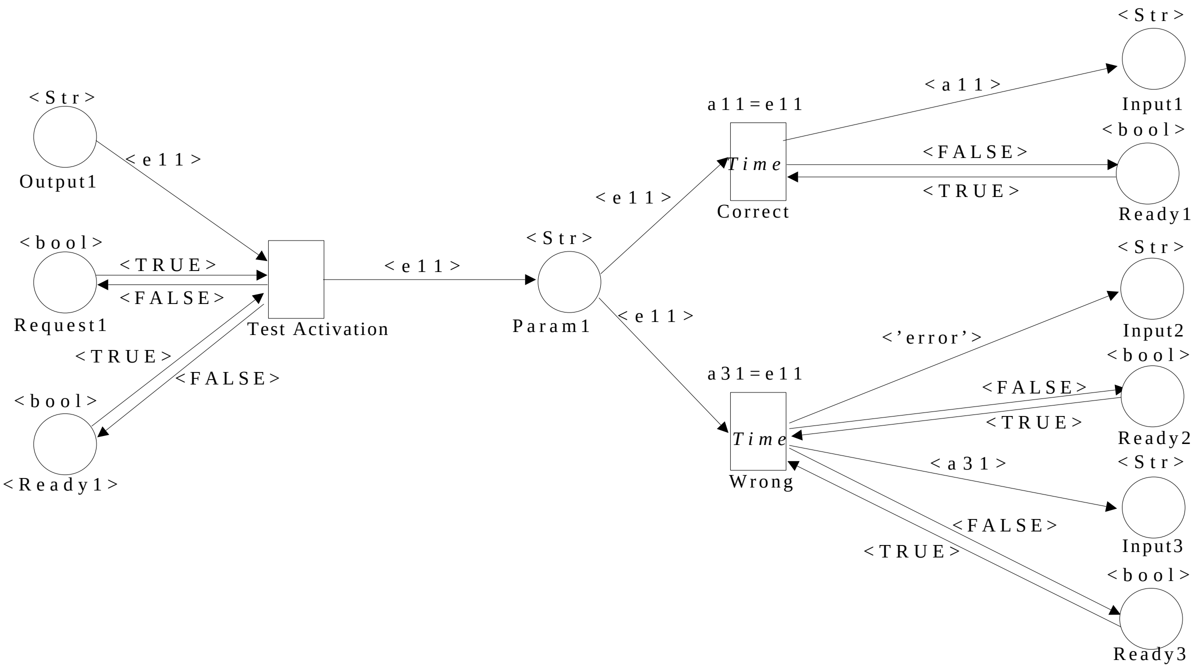


Figure 6: Pr/T net defining the job CheckDfd

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.

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.

In [Emme89] about 40 further jobs for software process modelling were defined in terms of Pr/T nets.

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 behaviour.

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 is the same as defined in [Ramc74]. 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.

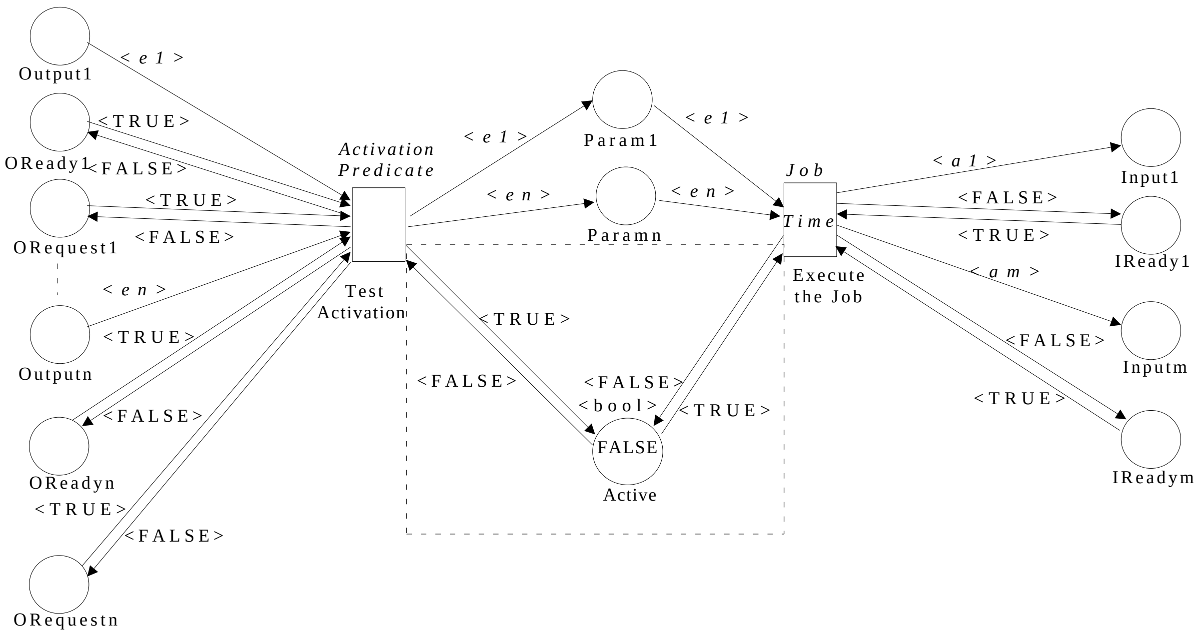


Figure 7: Extensions of a net defining a instance t with $A_W(t) = PIPE$

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.

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 described in [Emme89].

If the marking $M(s)$ is empty the initial marking of the unfolded channel corresponds to the

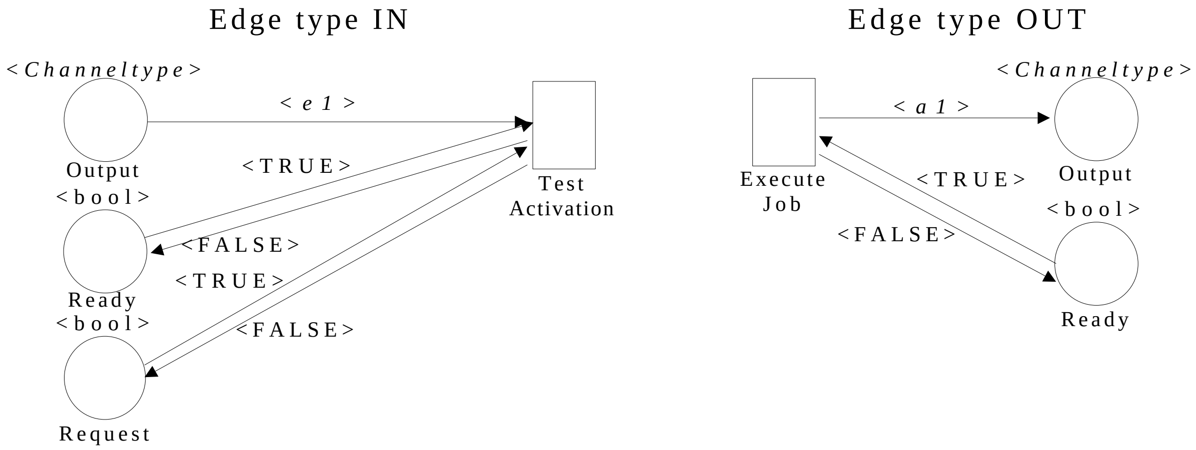


Figure 8: Pr/T net representation of edges with types IN and OU

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 $C_A(s) = FIFO$.

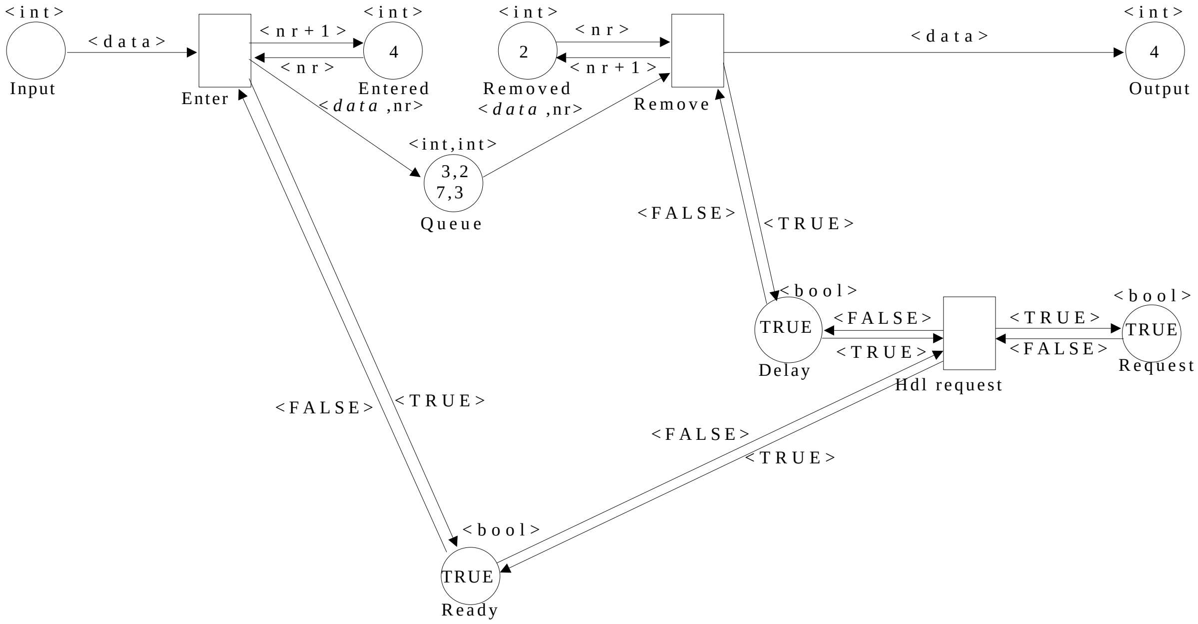


Figure 9: Translation of a Marking for $C_A = FIFO$

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*.

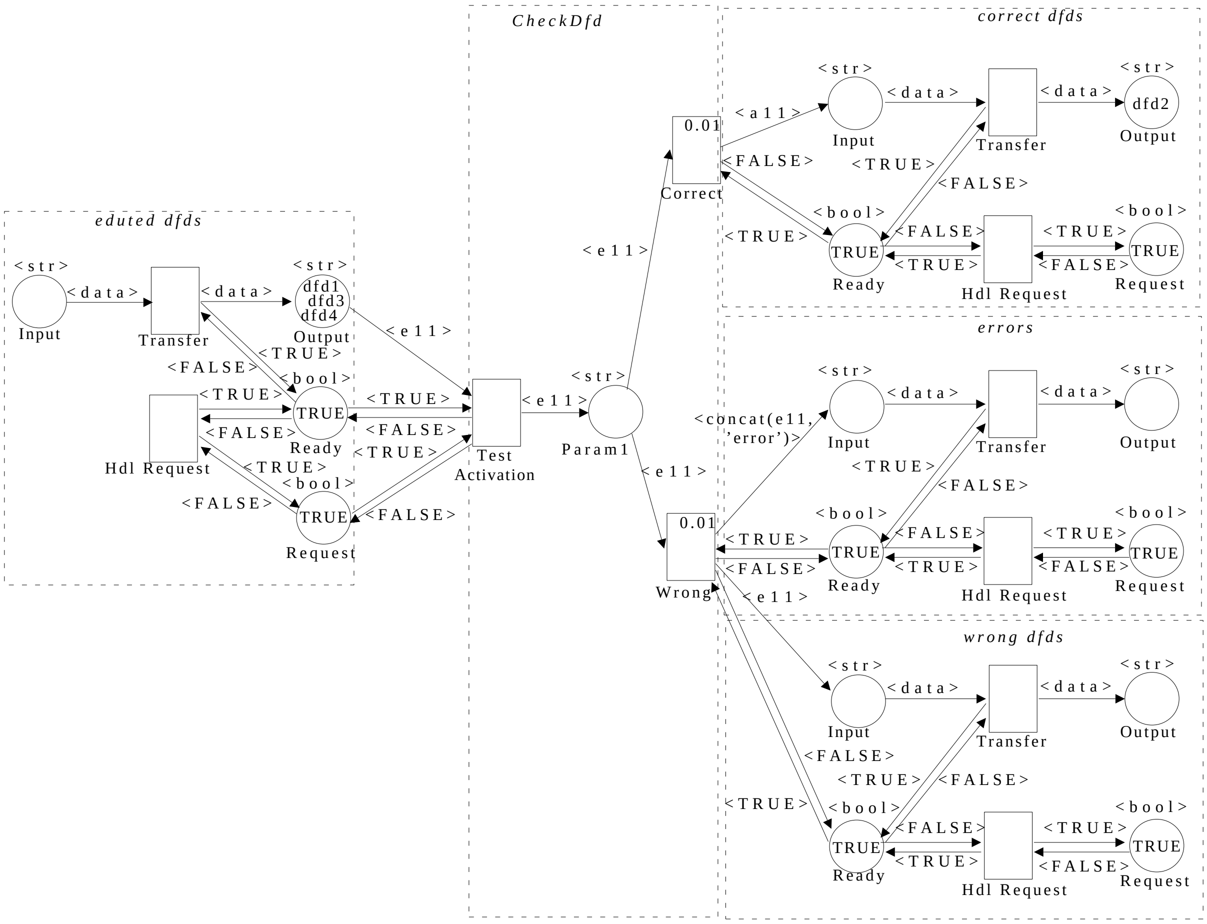


Figure 10: An assembled Pr/T net

3.6 Dynamic behaviour 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. Let $f(\bullet t \cup t)$ respectively $f(\bullet t \cup t \cup t\bullet)$ denote the unfolding of the instance t and its preset respectively pre- and postset together with the construction of their edges as described in subsection 3.5.7.

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.

Let $g(M)$ denote in the following the translation of the marking M by means of the algorithm sketched in subsection 3.5.8.

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 $f(\bullet t \cup t)$ is enabled under $g(M)$.

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 occurence is determined by the firing of* $t_1, \ldots, t_n \in f(\bullet t \cup t \cup t\bullet)$ *for which holds:*

$$g(M)[_{t_1:\alpha_1}\rangle \ldots [_{t_n:\alpha_n} g(M')$$

with $g(M')$ *enabling no transition of* $f(\bullet t \cup t \cup t\bullet)$

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 Analysis of FUNSOFT Nets

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

In principle we distinguish between validation of software process models (done by simulation) and verification of software process model properties. Validation is performed by using the FUNSOFT simulation tool which is described in detail in [MELM90]. In this section we focus on analyzing software process models by verifying software process model properties.

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.

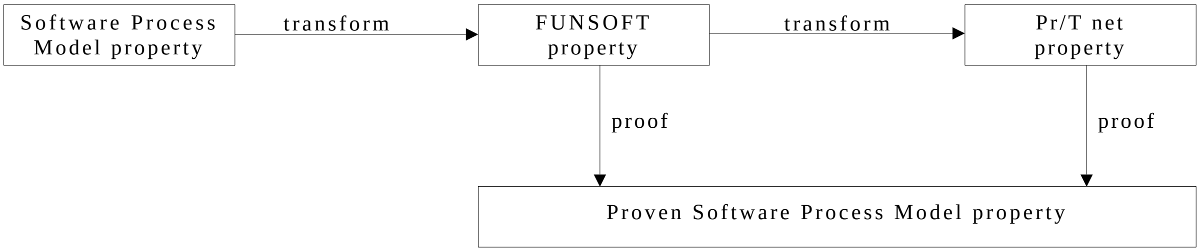


Figure 11: Verification method for FUNSOFT nets

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 modelling 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.

Properties concerning FUNSOFT node and edges attributes Properties of this class are proven by evaluating the attributes of channels, instances, and edges of FUNSOFT nets. One interesting software process model property is to find out if the described software processes require more than k programmers in order to be efficiently executed ($k \in I\!N$). Such a property can be checked by examining how many concurrently working instances are supported by programmers. Moreover, it can be shown that a certain percentage of edges between channels and instances are of type *CO*, which can be considered as a hint for a growing number of objects managed in the software process.

Structural properties of a FUNSOFT net Properties of this class are proven by structural analysis techniques. These analysis techniques are partially implemented by algorithms for detecting detect deadlocks, traps [Comm72], and conflicts [Reis86] and partially by the interpretation of S-invariants of the unfolded FUNSOFT net (for example for obtaining results concerning the conservativity [Pete81] of 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 and what obviously reveals an error in the software process model). By calculating S-invariants of the Pr/T net which results from unfolding the FUNSOFT net we were able to prove the strict conservativity of the FUNSOFT net depicted in Figure 2. For calculating S-invariants we used the tool described in [KL84] which is based on [Mevi81]. By means of this calculation we showed that no person disappears during the software process.

Dynamic properties of a FUNSOFT net Properties of this class are proven by dynamic analysis techniques. These techniques are implemented by algorithms which prove non-liveness [Laut73], fairness [Mura89], and the non-reachability of particular markings. These examinations are based on reachability trees. Reachability trees for arbitrary FUNSOFT nets are not finite. Thus we have to apply reduction mechanisms for reachability trees. The reduction we employed is the reduction to the number of tokens which abstracts from the individual value of tokens. This reduction corresponds to the total projection as introduced in [Genr86]. The reduction to the number of tokens means that we are not able to build reachability trees for FUNSOFT nets in which the values of tokens determine if a transition is enabled or not. Thus we are only able to consider simple FUNSOFT nets. Simple FUNSOFT nets are FUNSOFT nets in which no activation predicates are assigned to instances and in which no jobs with a MULT_IN or COMPLEX_IN input firing behaviour occur.

Thus we cannot obtain results as "it cannot occur a state in which the channel which contains modules is marked with the modules m1, m2, and m3" but we can obtain results which concern the mere quantitative aspects such as "it cannot occur a state in which the channel which contains modules is marked with three modules".

The mentioned properties are used for showing that a software process cannot reach final states (or that exactly this is possible), that conflicts between activities are resolved in a fair way. The non-reachability of particular markings corresponds to software process states which never can be reached.

Reverting to the net depicted in Figure 4 one is interested in examining if the described software processes reach final states (otherwise it cannot be guaranteed that the requirements phase ends at all) and if not more than one *sa-model* can exist at a certain point in time (since we have an inconsistent requirements analysis state otherwise). The reachability tree for the FUNSOFT net proves both properties by showing the k-boundedness of the channel encompassing *sa-models* and by identifying dead states.

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. Our current research focuses on removing the restriction on total projections of FUNSOFT nets (which is the source for only obtaining quantitative results). Our idea is not to use total projections, but to employ more sophisticated reduction methods as equivalent markings [Jens86] and stubborn sets [Valm89]. By using more sophisticated reduction methods we hope to avoid the restriction to simple FUNSOFT nets in the future.

5 Tool Support for FUNSOFT Nets

In this section we point out how our approach to the modelling of software processes and to the analysis of software process models is implemented. On the one hand this section gives a rough sketch of the relationships between some basic tools working with FUNSOFT nets and on the other hand some of these tools are discussed in a little more detail.

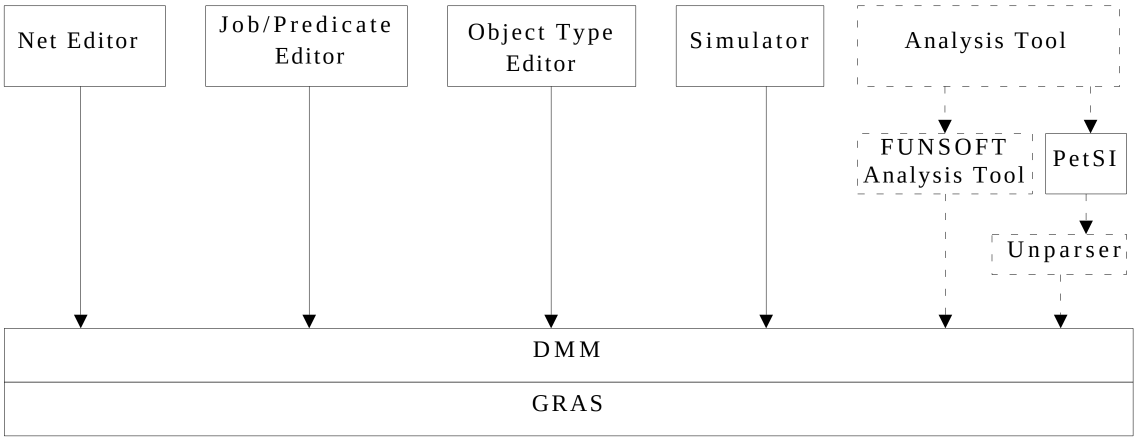


Figure 12: Relationships between basic FUNSOFT tools

Figure 12 provides a sketch of the relationships between the basic tools working with FUNSOFT nets. The arrows between different components represent the *use*-relationship between components. Components represented by dashed boxes are under implementation. The tools sketched in Figure 12 work in an incremental way, thus it is not necessary to predefine an order of their application.

The *Net Editor* is a graphic editor used for editing the skeleton of FUNSOFT nets. These nets are incrementally parsed and stored in the underlying *Object Management System*. The *Job/Predicate Editor* and the *Object Type Editor* are used for editing the components O, P, and J of a FUNSOFT net. These components are stored in the *Object Management System*, too. The other tools are retrieving the FUNSOFT net representation from this common storage medium. Since the *Analysis Tool* encompasses the tool *PetSI* it is necessary to build an *Unparser* which provides the required Pr/T format. *PetSI* is a tool for calculating S-invariants which is described in [Mevi81] and whose improvements are described in [KL84].

The *Simulator* accesses the object management system to get the information which are needed during simulation of FUNSOFT nets. In the FUNSOFT net simulation tool jobs and activation predicates implemented in the programming language C can be used.

The *Net Editor* enables the development of hierarchical FUNSOFT nets. This hierachical structuring is obtained by enabling the definition of instances which can be refined by subnets. This notion of refinement corresponds to the notion of substitution transitions as described in [HJS89]. Other notions of refinements are not implemented yet. According to the definition of hierarchical coloured Petri nets we consider the hierachical structuring as an operational feature which does not affect the semantics definition of FUNSOFT nets. The user interface of the editor is sketched in Figure 13.

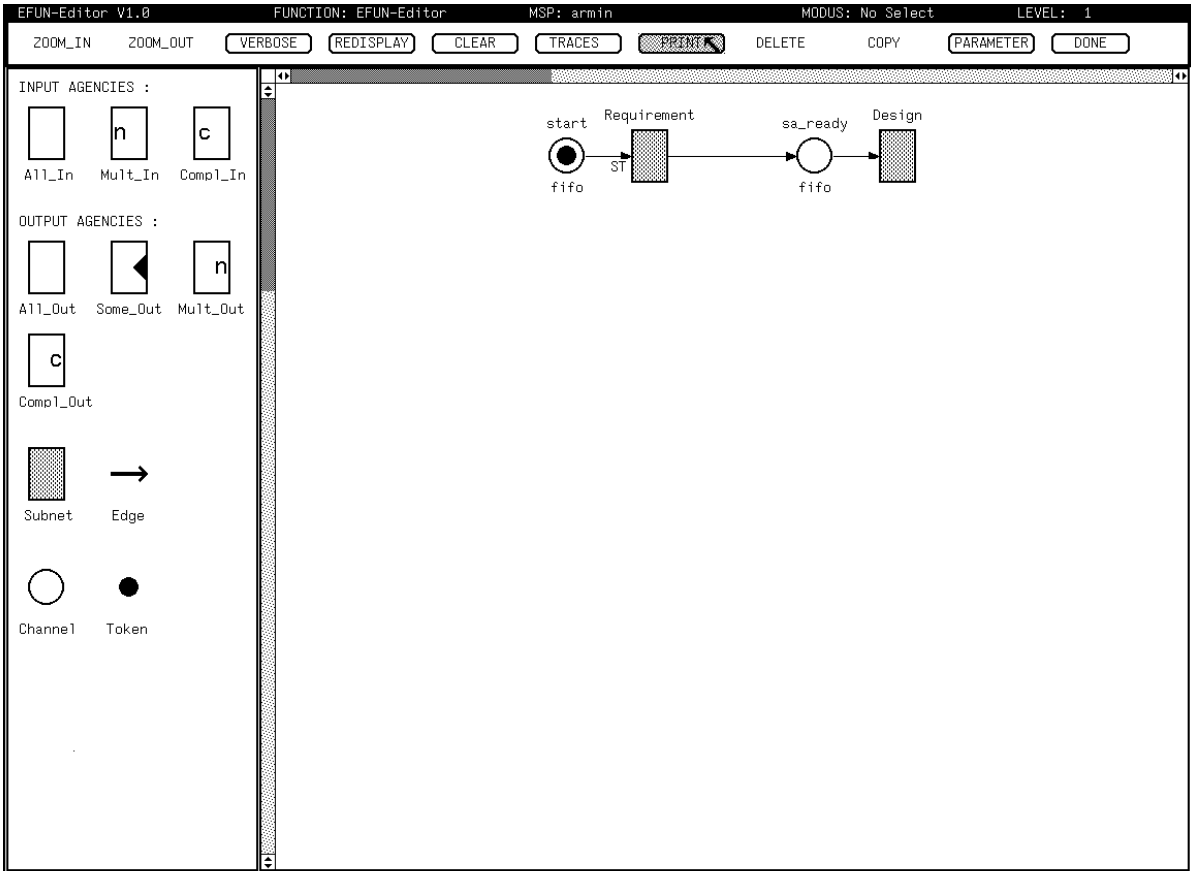


Figure 13: User interface of the Net editor

The *Analysis Tool* consists of three main components. Firstly, it has a FUNSOFT net analysis component which contains algorithms for directly examining FUNSOFT nets. Secondly, it has an unparser component which provides the equivalent Pr/T representation for the component which contains algorithms which are applied to the Pr/T net representation of the software process model. Thirdly, it encompasses the Pr/T net analysis tool *PetSI*.

The underlying object storage system is GRAS [LS88] which was developed in the IPSEN project [ELNS86]. GRAS is well suited for storing all kinds of graphs. The document management module (DMM) builds an application specific interface of GRAS. This module provides functions which enable the storage and the access of FUNSOFT components in a convenient way. Typical examples are functions for storing single edges, for retrieving attributes of particular nodes or for retrieving all input channels of an instance. By means of these function it is possible to access small parts of the FUNSOFT net. Thus, we are able to work with large FUNSOFT nets without creating copies in the main memory.

A more detailed description of the environment build around FUNSOFT nets can be found in [MELM90].

6 Conclusion

In this paper we described a result of combining knowledge in the area of software process modelling and in the area of Petri net research.

The result is an application oriented type of high-level Petri nets, namely FUNSOFT nets. FUNSOFT nets can be used in software process modelling and 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 described work is embedded in two European projects funded under the ESPRIT and the EUREKA programme. It was carried out under very pragmatic conditions, that means the suitability of the proposed formalism was an important argument throughout the whole development of FUNSOFT nets. The proposed type of high level nets has not only be proven to be suitable for describing software process models, it furthermore has contributed to measure up to some of the essential and often demanded requirements (namely analysis of software process models, graphic animation of software processes, and simulation of software processes). Thus, FUNSOFT nets are a reasonable candidate for a formal software process modelling language.

In the near future the implementation of an environment which enables to use FUNSOFT nets for modelling, simulating, and analysing of software processes and software process models will be finished. Our investigations of analysis techniques which are worthwhile to be used in software process modelling remain an ongoing activity. Especially the use of more sophisticated reduction mechanisms for FUNSOFT markings (instead of using total projections) seems to be promising with respect to increasing the expressive power of results delivered by standard Petri net analysis techniques.

Acknowledgements

A lot of ideas contained in this paper came up during intensive discussions with our colleagues in the ALF project. We are particularly indebted to our colleagues in Dortmund working in the cooperation between the chair for software technology and the STZ company, which is headed by W. Schäfer. These colleagues are W. Deiters, N. Madhavji, W. Stulken, B. Peuschel, H. Hünnekens, K.-J. Vagts, J. Cramer, and S. Wolf.

References

[11ICSE] *Proceedings of the 11^{th} International Conference on Software Engineering*; Pittsburgh, PA, USA; 1987

[2ISPW] *Proceedings of the 2^{nd} International Software Process Workshop*; Coto de Caza, CA, USA; 1986

[3ISPW] *Proceedings of the 3^{rd} International Software Process Workshop*; Beckenridge Colorado, USA; 1986

[4ISPW] *Proceedings of the 4^{th} International Software Process Workshop*; Moretonhampstead, Devon, UK; 1988

[5ISPW] *Proceedings of the 4^{th} International Software Process Workshop*; Kennebunkport, Main, USA; 1989

[9ICSE] *Proceedings of the 9^{th} International Conference on Software Engineering*; Monterey, CA, USA; 1987

[BB88] B. Boehm, F. Belz; *Applying process programming to the spiral model*; in: [4ISPW]

[BBCD89] K. Benali,N. Boudjlida, F. Charoy, J.-C. Derniame et.al.; *Presentation of the ALF project*; in: Proceedings of the International Conference on System Development Environments and Factories I; Pitman Publishing; London 1989

[BCG83] R. Balzer, T. Cheatham, C. Green; *Software Technology in the 1990's: Using a New Paradigm*; in: Computer, Vol. 16, No. 11, pp. 35-45; November 1983

[Boeh88] B. W. Boehm; *A Spiral Model of Software Development and Enhancement*; in: Computer May 1988; 1988

[Chro88] G. Chroust; *Application Development Project Support (ADPS)*; in: ACM SIGSOFT Software Engineering Notes, Vol. 14, No. 5, pp 83-104; July 1989

[Comm72] F. Commoner; *Deadlocks in Petri nets*; Applied Data Research Inc.; Wakefield, Mass, CA, 7206-2311; 1972

[DGS89] W. Deiters, V. Gruhn, W. Schäfer; *Systematic Development of Formal Software Process Models*; in: Proceedings of European Software Engineering Conference '89; LNCS 387; Springer; Berlin 1989

[DeMa79] T. de Marco; *Structured Analysis and System Specification*; ourdon Press; 1979

[Dows86a] M. Dowson; *The Structure of the Software Process*; in: [2ISPW]

[Dows86b] M. Dowson; *Workshop Introduction and Overview*; in: [3ISPW]

[Dows87] M. Dowson; *ISTAR and the Contractual Aproach*; in: [9ICSE]

[ELNS86] G. Engels, C. Lewerentz, M. Nagl, W. Schäfer; *On the Structure of an Incremental and Integrated Software Development Environment*; in: Proceedings of the 19^{th} Hawaii International Conference on System Sciences; 1986

[Emme89] W. Emmerich; *Semantik und Analyse von erweiterten Funktionsnetzen zur Unterstützung der Software-Prozeßmodellierung*; Master thesis; University of Dortmund; 1989

[Fink89] A. Finkelstein; *"Not Waving but Drowning": representation schemes for modelling software development*; in: [11ICSE]

[Genr86] H. J. Genrich; *Predicate/Transition Nets*; in: Petri Nets: Central Models and Their Properties; Lecture Notes in Computer Science, 254; Springer; Berlin 1986

[Godb83] H. P. Godbersen; *Funktionsnetze: Eine Modellierungskonzeption zur Entwurfs- und Entscheidungsunterstützung*; Ladewig-Verlag; Berlin 1983

[HJS89] P. Huber, K. Jensen, R. M. Shapiro; *Hierarchies in Coloured Petri Nets*; in: Proceedings of 10^{th} International Conference on Petri Nets; Bonn 1989

[HJPS89] H. Hünnekens, G. Junkermann, B. Peuschel, W. Schäfer, J. Vagts; *A Step Towards Knowledge-based Software Process Modeling*; in: Proceedings of the International Conference on System Development Environments and Factories I; Pitman Publishing; London 1989

[Jens86] K. Jensen; *Coloured Petri nets*; in: Petri Nets: Central Models and Their Properties; Lecture Notes in Computer Science, 254; Springer; Berlin 1986

[KF87] G. E. Kaiser, P. H. Feiler; *An Architecture for Intelligent Assistance in Software Development* in: [9ICSE]

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

[KR77] B. W. Kerninghan D. M. Ritchie; *The C Programming Language*; Prentice Hall; 1977

[Kell88] M. I. Kellner; *Representation formalisms for software process modelling* in: [4ISPW]

[Kell89] M. I. Kellner; *Software Process Modeling Experience*; in: [11ICSE]

[LS88] C. Lewerentz, A. Schürr; *GRAS – a Management System for Graph-like Documents*; in: Proceedings of the 3^{rd} International Converence on Data and Knowledge Bases, Jerusalem; Morgan Kaufmann Publishers Inc.;1988

[LST84] M. M. Lehman, V. Stenning, W. Turski;*Another Look at Software Design Methodology*; in: ACM SIGSOFT Software Engineering Notes Vol. 9, No. 2, pp 38-53;April 1984

[Laut73] K. Lautenbach; *Exakte Bedingungen der Lebendigkeit für eine Klasse von Petri Netzen*; GMD-Report No. 82; 1973

[Lehm87] M. M. Lehman; *Process Models, Process Programs, Programming Support*; in: [9ICSE]

[MELM90] Bröckers et.al.; *Endbericht der Projektgruppe Melmac*; Technical Report; University of Dortmund; to appear in 1990

[Mevi81] H. Mevissen; *Algebraische Bestimmung von S-Invarianten in Prädikat/Transitions-Netzen*; ISF-Report 81.01; Gesellschaft für Mathematik und Datenverarbeitung; Bonn 1981

[MGDS90] N. Madhavji, V. Gruhn, W. Deiters, W. Schäfer; *Prism = Methodology and Process-oriented Environment* to appear in: Proceedings of the 12^{th} International Conference on Software Engineering; Nice, 1990

[Mura89] T. Murata; *Petri Nets: Properties, Analysis and Applications* in: Proceedings of the IEEE, Vol. 77, No. 4, pp 541-578, April 1989

[Oste87] L .Osterweil; *Software Processes are Software Too*; in: [9ICSE]

[Pete81] J. L. Peterson; *Petri Net Theory and the Modelling of Systems*; Prentice-Hall; 1981

[Psi89] PSI GmbH; *Net Version 3.0*; PSI GmbH; Berlin 1989

[Ramc74] C. Ramchandani; *Analysis of asynchronous concurrent systems by timed Petri-Nets*; PH.D Thesis Massachusetts Institute of Technology; Cambridge, Massachusetts 1974

[Reis86] W. Reisig; *Petrinetze Eine Einführung*; 2. Auflage; Springer; Berlin 1986

[Robe88] C. Roberts; *Describing and Acting Process Models with PML*; in: [4ISPW]

[Royc70] W. W. Royce; *Managing the Development of Large Software Systems: Concepts and Techniques*; Proceedings, WESCON; 1970

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

[Star87] P. H. Starke; *On the mutual simulatability of different types of Petri nets*; in: Concurrency and Nets; Springer; Berlin 1987

[SW88] W. Schäfer, H. Weber; *The ESF-Profile*; Technical Report No. 242, Department of Computer Science, University of Dortmund; 1988

[TBCO88] R. N. Taylor, F. Belz, L. Clarke, L. Osterweil; *Foundations in the ARCADIA Environment*; in: Proceedings of the 3^{rd} Symposium on Software Development Environments; Boston, 1989

[Tayl86] R. N. Taylor; *Concurrency and Software Process Modells*; in: [3ISPW]

[Valm89] A. Valmari; *Stubborn Sets for Reduced State Space Generation*; in: Proceedings of 10^{th} International Conference on Applikation and Theory of Petri nets; Bonn 1989