A model for cooperative software design

ANTONIO CARLOS PEREIRA MAIA*
EDWARD HERMANN HAEUSLER
CARLOS JOSÉ PEREIRA DE LUCENA

Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro
E-mail: {maia, hermann, lucena@inf.puc-rio.br}
* (on leave from PETROBRÁS - Petróleo Brasileiro S.A.)

PUC-RioInf.MCC 09/96 March, 1996

ABSTRACT:

In this paper we present a cooperative software design process model. We are particularly concerned with software design as a social interaction activity. Consequently, we focus on cooperative design. By cooperative design we mean a form of design in which all information exchange and decision-making are collectively undertaken during design meetings. This sort of design is very interesting because it is strongly based on cooperation among people with various abilities and from different areas of expertise. This leads to a great diversity of cooperation (design) episodes to be analyzed. Design episodes have been previously represented using diagrams called Dynamics of Interaction Diagrams from which we have drawn up a specification of the cooperative design process. We have now used Coloured Petri Nets as the design process modeling language to produce a model based on this specification. We also discuss model validation and analyses results performed by a tool, called ANARCO, which considers three net properties: boundedness, liveness and re-initiation. Finally, we present our conclusions and provide comments on future work.

Keywords:

Cooperative software design, software design modeling, Coloured Petri Nets, formal specification.

RESUMO:

Neste trabalho apresentamos um modelo de processo de design cooperativo de software. Por estarmos interessados em design de software como uma atividade de interação social, nos concentramos em design cooperativo. Chamamos de design cooperativo uma forma de design na qual toda a troca de informações e tomada de decisões são realizadas coletivamente durante reuniões de design. Esta forma de design é particularmente interessante porque, por ser fortemente baseada em cooperação entre pessoas com habilidades distintas e especialidades diferentes, provê uma grande variedade de episódios de design para serem analisados. Episódios de design foram previamente representados em diagramas denominados Diagramas da Dinâmica das Interações, a partir dos quais derivamos uma especificação do processo de design cooperativo. Esta especificação foi usada como base para a geração do modelo de design empregando Redes de Petri Coloridas como a linguagem de modelagem do processo. Utilizamos uma ferramenta, denominada ANARCO, para análise e validação do modelo e que considera três propriedades de redes: limitação, vivacidade e reinício. Finalmente, apresentamos conclusões e sugestões para trabalho futuro.

Palavras-Chave:

Design cooperativo de software, modelagem de design de software, Redes de Petri Coloridas, especificação formal.

1. Introduction

In recent years, important research efforts have been devoted to characterizing, assessing, and improving complex business and engineering processes distinguished by great emphasis on information exchange, partial automation (since many trade-offs and decisions must be accomplished by humans and cannot be automated) and cooperation among humans (Fuggetta and Ghezzi 1994). They have been called human-oriented or human-centered systems (Conradi et all 1993). Typical examples are engineering processes, such as software design, office automation processes and MIS processes.

These research efforts have led to the development of an area of research known as software process modeling and technology (Finkelstein et al 1994). This area is particularly concerned with the development of process-centered (sensitive) software engineering environments (PSEEs). That is, computer based systems providing a variety of facilities working together to support people who develop software (Snowdon and Warboys 1994). In these environments there is an explicit definition of the process to be followed during software development. This definition, called the process model, can be used to describe, validate, and communicate the process(Fuggetta and Ghezzi 1994).

A number of these environments are proposed in the literature. They are based on process-modeling languages (PMLs) that are used to describe the process model. These languages are formally based on a variety of different paradigms (Finkelstein et al 1994), including active data bases, artificial intelligence rules, graph/nets and process programming (Fuggetta and Ghezzi 1994). Examples of environments/tools exploiting the graph/net paradigm are: MELMAC / MSP (Deiters et al 1989) and SPADE (Bandinelli and Fuggetta 1993). They use different kinds of high-level Petri nets to model software processes.

In our research we are particularly concerned with the identification of requirements for a computational environment to support the cooperative software design process. In order to achieve this research goal we also took a process-centered approach. This environment, like a PSEE, must be based on a process model whose requirements must be captured before it can be designed and developed. In a previous work (Maia, Garcia and Lucena 1995) we have proposed a methodology to support the elicitation of requirements to support cooperative software design process. In another work (Maia, Fuks and Lucena 1995) we introduced the specification of a cooperative design process model. This specification consists of a number of properties which comprise temporal relations between speech objects (representations of verbal communication actions) and the conditions for and constraints to enabling the utterance of specific speech objects. In this current work we introduce the resulting cooperative design process model using a high-level Petri net - Coloured Petri nets - as our process modeling language.

The use of high-level Petri nets as a notation for specifying group and man-machine interaction is also reported in the literature. In (Furuta and Stotts1994) collaboration protocols and their use in groupware prototyping are proposed. An interpretation of a Coloured Timed Net (CTN), the net notation used in a system named Trellis, as a collaboration protocol is presented. The net is annotated with fragments of information (text, graphics, video, etc.) and allows event timing specification. In (Holt 1988) a PT-net-based graphical specification language was designed for coordination of

multiple cooperating agents in an information processing organization. Finally, in (van Biljon 1988) Petri nets are used for specifying man-machines dialogues.

2. Petri Net-Based Models of Software Engineering Processes

Some process modeling efforts have been based on languages derived from Petri nets. MELMAC (Deiters and Gruhn 1990), for example, is an environment which supports the use of software process models and represents them by FUNSOFT nets. FUNSOFT nets (Gruhn and Jegelka 1992) are a high-level type of Petri net whose semantics is defined in terms of Pr/Tr nets extended by multi-sets (Genrich 1986). SPADE (Bandinelli and Fuggetta 1993), developed by Politecnico de Milano and CEFRIEL, is another environment which is based on a process modeling language, called SLANG. SLANG has been formally based on a high-level Petri net formalism, called Environment/Relationship (ER) nets (Ghezzi et al 1991).

Software process models specify allowed schedules of interacting software development activities that are carried out in parallel by a number of cooperating agents. These activities must also meet time constraints. In MELMAC these activities are described by small programs or they are determined by the invocation of tools. Additionally, activities can also depend on human thinking and problem-solving. But, this sort of activity is merely described in terms of the types of input and output objects. In SPADE, activities are represented by transitions. A transition is associated with an action which either specifies a computation on incoming data types or asynchronously invokes a non-SLANG executable routine. Examples of activities represented in these environments are: test-a-system, compile and change-design.

In (Krämer and Luqi 1991) a Petri net-based specification method for distributed systems is applied to the domain of software process modeling. This formal approach is used to specify the dynamic behavior of software engineering processes, characteristic attributes of software objects, and tasks of human participants involved. Nevertheless, like previously commented projects, it does not address human factors and social processes.

As pointed out in (Bandinelli et al 1994) these process modeling languages were designed to be used for process modeling-in-the-large. They represent activities that may be classified as automatic (that can be carried out without human interaction) and interactive (that require the interaction of a single person with the environment). But, there are several activities, called social interaction activities (Gruhn 1992), that require interactions between human agents. Examples are requirement definition with customers, change request evaluation and cooperative design.

In our research work we focus on cooperative design. By cooperative design we mean a form of design where all information exchange and decision-making are collectively undertaken during design meetings. In contrast to previous approaches, we are concerned with software design as a social interaction activity. We also focus on the design process and not on the whole software development process. We propose the use of a concurrent automaton, Coloured Petri nets, for (design) process modeling in-the-small. Design episodes are described by verbal and non-verbal communication actions undertaken by cooperating members of a design team.

In the next section we briefly introduce a method, called the Dynamics of Interaction Method, for analyzing audiovisual records of teamwork design and the elicitation of requirements to support cooperative software design process.

3. A Method for Analyzing Team Design Activity

3.1 DESIGN ACTIVITY OBSERVATION

In our research work we are particularly interested in the elicitation of requirements to support cooperative software design process. To reach this research goal we first videotaped and then analyzed the design activity of a multi-disciplinary group from between five to eight people. The team designed an educational software system. The design team was composed of up to seven people: a Software Designer, an Education Expert, a Domain Expert and four users' representatives who were responsible for bringing the students needs from the perspective of each professional area involved (business administration, quality control, geology and industrial engineering).

In the literature, there are many methods for observing the design activity. These methods vary from simple documentation analysis to video tape analysis. Although video tape technology was previously used to record the design activity, until recently systematic methods had not been developed to analyze video data. For more information on this recent development see (Maia, Garcia and Lucena 1995) which presents in details a representational scheme for such analysis of team design activity recorded on videotape. The representational scheme focused on the dynamics of interaction between people involved in design.

3.2 DYNAMICS OF INTERACTION

In this section we briefly introduce a method, called the Dynamics of Interaction Method, for analyzing audiovisual records of teamwork design. The method leads to a representational scheme which focuses on the dynamics of interaction between team members. A thorough description of the method can be found in a previous paper (Maia, Garcia and Lucena 1995).

The handling of the information produced during design meetings is the kernel of the method. In cooperative design all information exchange is performed during meetings. The whole design is based on collective decisions. Individual activities outside the meetings are restricted to personal documentation, recording of the discussion from a meeting and preparation of materials of various kinds to be submitted for consideration to the design team. Although this particular kind of design is not so common in current design practice, it is very interesting and appropriate for our research goals because it is strongly based on cooperation between people with various abilities and from different areas of expertise which leads to a great diversity of cooperation episodes to be analyzed.

Our first step towards the creation of the representational scheme was to define a language that would show the dynamics of interaction between team members. This language is used to express visually relevant sequences of the design practice, abstracting from the specifics of the application domain by providing a representation that reveals communication actions (both verbal and non-verbal ones- such as gestures and looks). A list of some communication actions are presented in Appendix A.

In order to provide a visual organization for the new description of the design activity we proposed to display communication actions in diagrams - we have called them Dynamics of Interaction Diagrams. These diagrams preserve the temporal relations and parallelism among communication actions. We also wanted to register turn-taking

allocation. The design episode description should identify turn transitions and indicate which turn allocation mechanism has been applied, not only verbal ones (by naming the person to take the turn) but also non-verbal ones (staring at the person to whom you want to address a query).

We claim that the use of concurrent automata to represent team design is possible due to its reactive nature. A reactive system was defined by (Harel 1988) as being characterized by its event driven nature, continuously reacting to stimuli. The behavior of a reactive system is really the set of allowed sequences of input and output events, conditions, actions and information flow. Team design behavior is also a set of permitted sequences of input and output events, particularly, communication actions. We may characterize an interaction state by the relationship between its input or incoming communication actions and its permitted output communication actions.

Dynamics of Interaction Diagrams represent parallel sequences of input and output communication actions that actually took place during design meetings. We studied these diagrams, looking for sets of input/output for each particular input communication action. As a result we derived for each input event (communication action) a fixed set of permitted output communication actions. From this we drew up a model of the reactive behavior of the team design activity and identified several temporal relations between communication actions. Finally, we have derived a specification of the design process expressed in terms of properties. For example, the first two properties deal with the initiation of a design episode, i.e., how designers become engaged in a design task:

I. At the beginning of every design episode communications actions **i** , **<i** and **q** must be available.

II. Only the communication actions **i** , **<i** and **q** are available at the beginning of every design episode.

These two properties state that the beginning of a design episode is determined by performing one of the listed communication actions: **i** (to raise a new issue), **<i** (to raise a former issue) or **q** (to make a query). Other properties are listed in APPENDIX B.

4. Cooperative Design Process

In the next sub-sections we introduce a group interaction protocol and a design problem solving process model that uses the group interaction protocol for cooperation.

The group interaction protocol describes the communicative behavior of designers during design meetings while the design problem-solving process model describes design episodes. The problem-solving process model presents the communicative actions taken by designers for attending a problem-solving or information request. Although the group interaction protocol and the design problem-solving process model compose a single model - a cooperative design process model - they are represented separately for the sake of simplicity. Since they model different entities (the former models human agents and turn-taking while the latter models design episode status, issues, queries and alternative values) we can think of them as parallel and complementary processes. In terms of computational systems to support cooperative design implementing them as separate modules would lead to a modular organization that facilitates maintenance and reuse.

4.1 A GROUP INTERACTION PROTOCOL

In this section we present a group interaction protocol and use it to introduce basic concepts of Petri nets and, in particular, of Coloured Petri Nets - CP-nets.

Petri nets are a specification language which is state and action oriented at the same time. The states of a CP-net are represented by places (which are drawn as circles or ellipses). Actions are represented by transitions (which are drawn as bars).

In a cooperative design meeting there are a number of agents (participants, designers) taking part in the design process. In order to speak an agent must get the turn. We assume, based on videotape data of design meetings, that (most of the time) two agents do not speak in parallel. However, an agent may , for example, supply a verbal feedback (such as uttering 'Yes' or 'No') while another agent is speaking.

Figure 1 shows a CP-net that represents a group interaction protocol in a cooperative design meeting. There are five different places. Three of them represent the three possible states of the agents: IDLE, SPEAK and WRITE. Two of the places indicate whether a turn is taken or available: TAKEN and AVAILB.

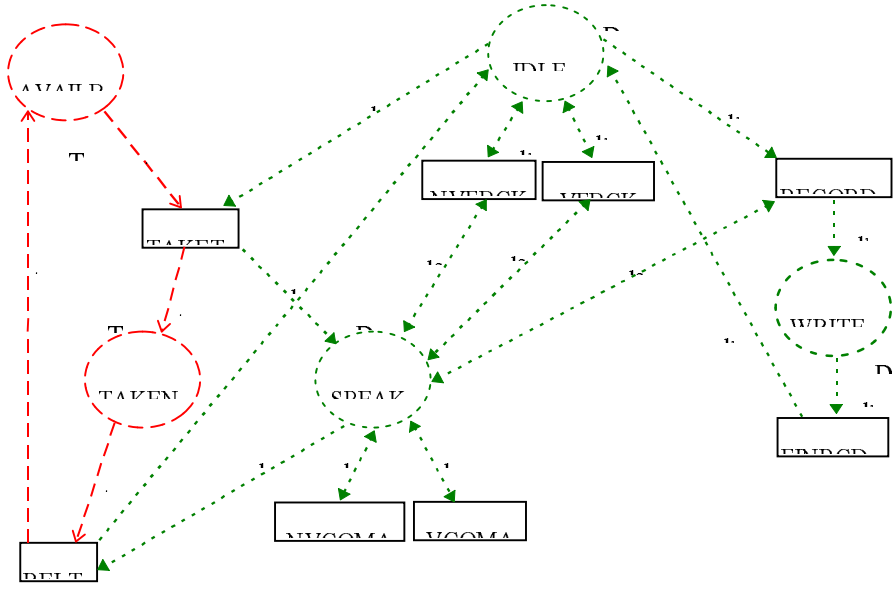

Figure 1 A group interaction protocol

Each place has an associated data type determining the type of data which the place may contain. Each place may contain a number of tokens. Tokens represent activities. For example, one token of the type (colour) D (which stands for a class of designers) in the place SPEAK indicates that an agent is speaking.

A state of a CP-net is called a marking. A marking is a function which maps each place into a multi-set of tokens of the correct type.

Each token carries a data value which belongs to the type of the corresponding place. IDLE, SPEAK and WRITE have the type:

D = {SD, EE, DE, GR}

where SD, EE, DE and GR are agents identifiers (which stand for Software Designer, Education Expert, Domain Expert and Geology's Representative). We assume a design meeting with four participants.

Execution of a Petri net begins with some initial marking which is specified by means of initialization expressions (not indicated in figure 1). For the places of type D we have the following initial marking:

M^{o} (IDLE) = 1*SD + 1*EE + 1*DE + 1*GR

M^{o} (SPEAK) = M^{o} (WRITE) = ϕ

where ϕ denotes the empty multi-set and the integer coefficients indicate that IDLE has one token for each data value in the type D.

TAKEN and AVAILB have the type T = {t} which has only one possible value. For the places of the type T we have the following initial marking:

M^{o} (AVAILB) = 1*t

M^{o} (TAKEN) = ϕ

Once an initial marking is specified, execution of a Petri net continues through a sequence of state changes caused by choosing an enabled transition in the current state and firing it to get a new state (marking). There are eight different transitions in figure 1. An incoming arc indicates that the transition may remove tokens from the corresponding place while an outgoing arc indicates that the transition may add tokens. The number of tokens removed from or added to a place when the transition occurs as well as their data values depend on the evaluation of arc expressions. The transition TAKET has two incoming and two outgoing arcs with two different arc expressions: t and d. The former is a constant while the latter contains the free variable d of the type D. To evaluate the arc expression we need to bind d to a value from D.

A transition is enabled if there is one or more bindings for the expressions. Let us consider the initial marking (as specified above). The transition TAKET may occur with the following bindings <d=SD>, <d=EE>, <d=DE>, <d=GR>. If we assume that the transition TAKET occurs with the binding <d=SD>, it will remove a token with value t from AVAILB and a token with value SD from IDLE and will add a token with value t to TAKEN and a token with value SD to SPEAK. This is interpreted as an agent taking the turn and speaking.

The occurrence of the transition TAKET promotes a state change represented by a new marking M^{1} with the following tokens:

M^{1} (IDLE) = 1*EE + 1*DE + 1*GR

M^{1} (SPEAK) = 1*SD

M^{1} (TAKEN) = 1*t

In the marking M^{1} five transitions are enabled: VCOMA, NVCOMA, VFBCK, NVFBCK and RECORD. The first two transitions are enabled with the binding <d=SD>. These transitions are interpreted as a Verbal COMmunication Action and a Non-Verbal COMmunication Action respectively. This means that while an agent is speaking (and, consequently, holding the turn) he/she may perform verbal (such as Supply rationale or Raise an issue) and non-verbal communication actions (such as, for example, he/she may point to an object or to another agent). In the next section we introduce the design problem solving process model where these communication actions are explicitly represented.

While SD is speaking, other agents may supply verbal feedback (transition VFBCK) or non-verbal feedback (transition NVFBCK) such as indicating agreement to what is being said by uttering 'Yes' or nodding ones head. The last transition RECORD represents a common situation when one or more agents take notes (or make drawings) of what is being proposed by the agent who is speaking. Since these transitions can occur with different tokens (for example, VCOMA with <d=SD>, NVFBCK with <d=EE> and RECORD with <d=DE> and <d=GR>) they do not influence each other. This means that they can occur concurrently with each other in the same step.
In the next sub-section we introduce the design problem-solving process model. As we have already introduced basics concepts of CP-nets, now we focus on the model.

4.2 DESIGN PROBLEM SOLVING PROCESS

During a design meeting, designers are engaged in a cooperative design task (we call it a design episode) or are performing another meeting activity, for example, planning the meeting itself, or they are simply engaged in side digressions (not taking part in a design-episode). In this work we are particularly interested in what happens within a design episode since it is where design problem-solving takes place. We consider that designers are taking part in a design-episode when they are attending a problem-solving or information (related to the problem domain) request.
Design episodes are represented by the verbal and non-verbal communication actions undertaken by designers until the information request (make a query) is attended or the problem-solving request (raise an issue) is solved.
In the next sub-sections we introduce the proposed model. Not the whole model at once, but, for the sake of simplicity, we introduce different sections of it and, whenever necessary, we refine a net section in order to show the model in different levels of details.

4.2.1 Design Episode Initiation

In figure 2 we present a section of the model which represents the beginning of a design episode.
This net section contains seven different places. Two of these places, INACTV and ACTIVE, represent two possible states of a design episode : inactive and active, respectively. These places are of the data type DE which has only one possible value DE = {detokn} - a constant used to denote a design episode unit/instance.
Two of them, UNUSED and FOCUS, have the type:

Trgt = {NEWI, FORI, QRY}

where NEWI represents that the target of that design episode is a new issue,
FORI represents that the target of that design episode is a former issue, and
QRY represents that the target of that design episode is a query.

Two of them, RAISED and SOLVED, have the type:

Issue = {I1, I2, ...In}

where I1 ... In represent identifiers of the issues under consideration.

Finally, UNSDQ and MADE, have the type:

Query = {PRIMQR, SECDQR}

where PRIMQR represents the identifier of a query which initiated the design episode, and

SECDQR represents a identifier of any subsequent query.

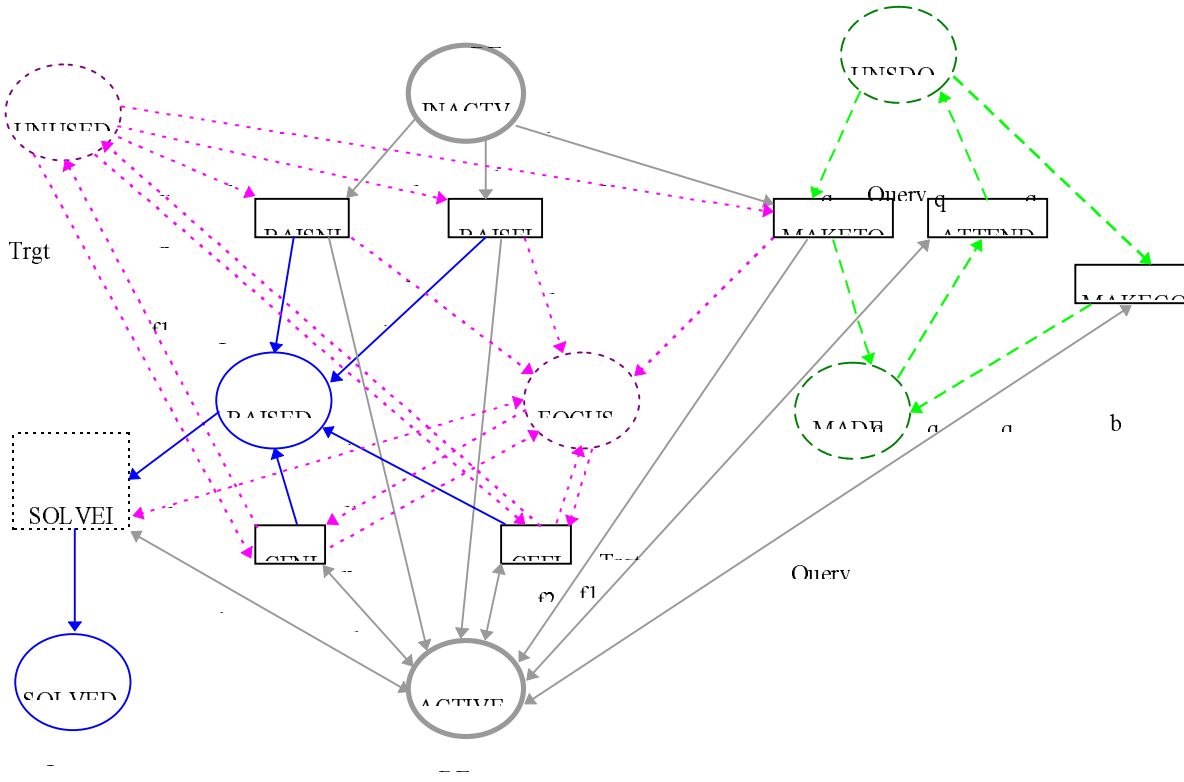

Figure 2. Design Episode Initiation

For the places of this net section we have the following initial marking:

M_0 (INACTV) = 1*detokn

M_0 (UNUSED) = 1*NEWI + 1*FORI + 1*QRY

M_0 (UNSDQ) = 1*PRIMQR + 1*SECDQR

All other places have the empty multi-set in the initial marking.

Designers start a design episode by requesting for problem-solving (raising an issue) or requesting for information (making a query). When requesting problem-solving there are two possibilities: a new issue (not solved) is raised or a former issue (already solved in a previous occasion) is revisited. These events are represented by the transitions RAISNI and RAISFI. Hence, to represent the beginning of a design episode there are three transitions available: RAISNI (to raise a new issue), RAISFI (to raise a former issue) and MAKETQ (to make a query). All of them remove a token with value detokn from INACTV and add a token with value detokn to ACTIVE. These three transitions are the only ones enabled in the initial marking. The choice between them is non-deterministic. As soon as one of them has occurred, the others become disabled - because there is no longer a token on INACTV. When this sort of situation takes place, we say that the corresponding transitions are in conflict with each other.

Additionally, the transition RAISNI removes a token with value NEWI from UNUSED and adds it to FOCUS. This is interpreted as being a new issue the current focus of the design episode. In a similar way, transitions RAISFI and MAKETQ remove a token with value FORI and QRY, respectively, from UNUSED and add the corresponding token to FOCUS. Consequently, these transitions establish that a former issue or a query, respectively, is the current focus of the design episode.

Transitions RAISNI and RAISFI also add a token with a identifier/value I1 to RAISED. This means that an issue was raised and is now under consideration by the design team. On the other hand, transition MAKETQ takes a token with a identifier PRIMQY from UNSDQ and adds it to MADE. This means that a query was made and should be attended by the design team. Subsequent queries are represented by the transition MAKEGQ which removes a token with value SECDQR from UNSDQ and adds it to MADE. The transition ATTEND represents the action of attending a query by any means (for example, supplying a rationale or an agreement).

There are situations during design episodes in which a request for information may introduce the discussion of an issue, i.e., call the team's attention to the fact that a particular issue should be considered/solved. It may be a new issue or an issue already discussed beforehand (former issue). The issue is brought under consideration as a consequence of the query. This is modeled by the two transitions CFNI and CFFI. The transition CFNI exchanges the current focus from query to new issue and transition CFFI exchanges it from query to former issue. Both of them remove a token with value QRY from FOCUS and add it back to UNUSED and also add a token with a identifier/value I1 to RAISED. Additionally, transition CFNI (CFFI) removes a token with value NEWI (FORI) from UNUSED and add it to FOCUS.

In figure 2 we represent a transition SOLVEI in a broken-line box-like enclosure. This particular notation is to indicate a net component that will be further refined. We refine a net by substituting an entire net for a transition, as in the case, or a place. This transition replaces a net which describes the problem-solving process that is triggered by the raising of an issue. This process will be showed in a following sub-section. Anyway, this transition takes a token of the type issue from RAISED and adds it to SOLVED. This means that the issue went into consideration by the design team and a decision was made, that is, an alternative value was assigned to this issue.

4.2.2 Design Episode Termination

Transitions STOP1 and STOP2, figure 3, represent the end of a design episode. There are two possible situations. First, the design team made a decision and, consequently, the issue under consideration was solved. Consequently, there is a token of the type Issue in SOLVED. Second, the design team did not reach a consensus or due to time constraints, the team decided to postpone that issue to another meeting, or by whatever reason decided to quit that design episode. Consequently, the raised issue was not solved. Hence, there is a token of the type Issue in RAISED.

Transition STOP1 removes a token of type Issue from SOLVED, a token of value detokn from ACTIVE and a token of type Trgt from FOCUS and adds a token of value detokn to INACTV and a token of type Trgt to UNUSED. In a similar way, transition STOP2 removes a token of type Issue from RAISED, a token of value detokn from ACTIVE and a token of type Trgt from FOCUS and adds a token of value detokn to INACTV and a token of type Trgt to UNUSED. Both transitions remove and put back the two queries identifiers from UNSDQ. This is to check if queries made were attended.

The occurrence of these transitions represents the end of a design episode. The place INACTV has again a token of value detokn which enables the beginning of a new design episode.

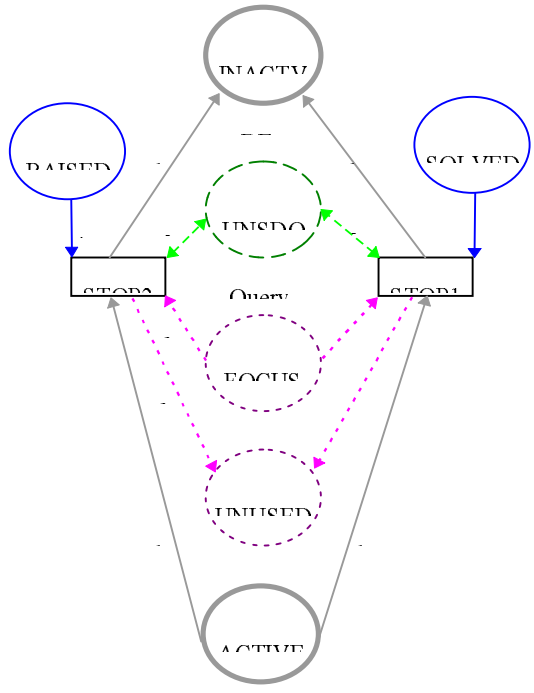

Figure 3. Design Episode Termination

4.2.3 Dependent Issues

Design is a pragmatic activity. Empirical studies (Curtis 1992) show that good designers tend to use a mixture of top-down, bottom-up, and middle-out analytical and synthetic approaches. Designers tend to adopt an opportunistic approach. For example, there are situations during design episodes in which a designer may propose the discussion of a new issue strongly related to the current issue (sometimes a generalization or a specialization of the current one). This situation usually takes place when an issue is raised but the team perceives that they first need to solve another issue upon which the current one is dependent, i.e., there is dependence relation between the issues. This is represented in figure 4 by the transition NEST. After discussing the nested issue, the team may return to the previous context, which is represented by the transition RESUME. Nevertheless, when discussing the nested issue a designer may propose a new nested issue. This case is represented by the transition NESTA. This situation leads to a chain of nested issues or a dependence net (alternative values for dependent issues are assigned after the issues on which they depend are solved). In this case, the transition RESUME will always take it to the previous (dependent) context.

The occurrence of the transition NEST takes a token of value detokn from ACTIVE and adds it to NESTED. This represents the fact that a nested issue is under consideration. This transition also adds a token of the type PairI to NESTDI. Actually, PairI is a pair of issues composed of the former and the nested one (in this order). A posteriori, this pair of issues will eventually be removed from NESTDI either by the transition RESUME or by the transition EMPTY.

When discussing a nested-issue the team may want to give up the whole reasoning line (formed by a number of nested issues) and return to the context of the first issue. This is made possible in the model by the transition DISCRI. The transition DISCRI is enabled if there is a token of value detokn in NESTED. This transition adds a token of value detokn to WAIT. If a token of value detokn is in WAIT only two transitions are possibly enabled: EMPTY and

RSTORE. The transition EMPTY removes tokens of type PairI from NESTDI. When all tokens are removed from NESTDI, there will be a token of the same value in RAISED and in PRIMRY. This additional pre-condition allows the occurrence of the transition RSTORE which restores the initial context, that is, the context of the first issue.

We can interpret the aforementioned transitions as implementing a stack or LIFO (Last In, First Out) structure. NESTDI stores a stack of nested issues while PRIMRY stores the first issue and RAISED stores the last (current) issue. Transitions NEST and NESTA insert the first and further pairs of issues in the stack respectively. On the other hand, transition RESUME removes the top of the stack. So does the transition EMPTY but on a different and specific context. The former is enabled when there is a token of value detokn in NESTED and represents the designer intent of resuming the previous context while the latter is enabled when there is a token of value detokn in WAIT and represents the designer intent of giving up a reasoning line (and consequently, emptying the stack). Finally, RSTORE checks if the stack is empty and resets the initial context inserting a token of value detokn is in ACTIVE.

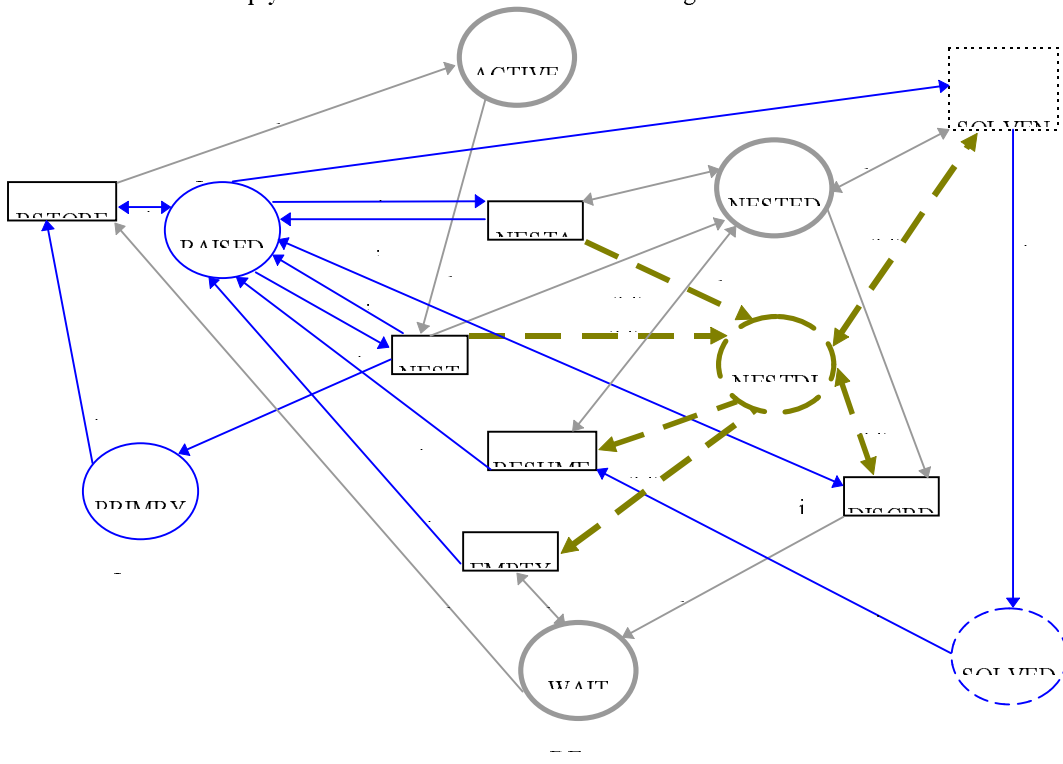

Figure 4. A net section representing the process of nesting issues

The transition SOLVEN (in figure 4) is similar to the transition SOLVEI (in figure 2). It takes a token of the type issue from RAISED and adds it to SOLVED. This means that a nested issue went into consideration by the design team and a decision was made, that is, an alternative value was assigned to this nested issue. However, SOLVEN is only enabled if there is a token of value detokn in NESTED while SOLVEI is enabled if there is a token of value detokn in ACTIVE. These transitions represent equivalent design problem-solving processes.

4.2.4 Problem-Solving

Problem-solving is a major activity in design. It involves the generation, refinement and evaluation of the designers' conceptualization of a solution, figure 5. When requesting problem-solving there are two possibilities: a new issue (not solved) is raised or a former issue (already solved in a previous occasion) is revisited.

If an issue is being revisited, it brings the alternative value assigned to it when he was first solved - that is, the decision made. This is represented by the transition RECVFD which recovers the former decision. This transition adds a token of type AV to the place GENRTD. This place holds generated alternative values. An alternative value is represented by one identifier.

Hence, the place GENRTD has the type:

AV = {AV1, AV2, ...AVn}

where AV1 ... AVn represent identifiers of alternative values.

Once one issue is raised or revisited designers supply alternative values (generate possible solutions) to this issue. This is represented by the transition GENER which also adds tokens of the type AV to GENRTD.

Alternative values assigned to an issue under consideration are liable to elaboration, that is, designers may suggest that features should be added or subtracted from the original alternative value. We assume that features added or subtracted from alternative values actually generate new alternative values which are considered by their own. Consequently, this elaboration process is also represented by the transition GENER.

After or even concomitant with the generation of solutions, designers evaluate proposed solutions expressing their agreement or disagreement to one or another solution, supplying rationale to support their positions or just expressing their preference for a particular solution. These communication actions are represented by the transitions EVAL1 and EVAL2. There are two similar transitions to represent these events because there are two different pre-conditions. EVAL1 takes a token of type AV from GENRTD while EVAL2 takes a token of type AV from EVALTD. This represents the fact that the same alternative value may suffer different evaluations by different designers.

Besides generating and evaluating possible solutions, designers also have to select one of them to be assigned to the design parameter (issue) under consideration. After discussing one or more alternative values the team should converge to one of them. We have observed two common strategies normally applied when designers feel that they are reaching a consensus in relation to one of the proposed solutions (alternative values): verification and confirmation. The former is applied when a designer wants to check if another designer (who has not yet committed to the candidate alternative value) agrees to this alternative value. This situation is represented by a transition DEMAGR (Demand agreement). It takes a token of type AV from EVALTD and adds it to CANDDT, which means that there is a candidate alternative value to be assigned to the issue. This transition also takes a token of value dectkn from NOTSLD and adds it to SLTD. This represents the fact that there is a possible decision been considered. Designers can express agreement to the candidate alternative value which is represented by the transition AGREE. This transition takes a token of type AV from CANDDT and adds it to SELECT. SELECT holds the token which identifies the decision made. Once one alternative value is selected, designers are likely to ask for its confirmation. This is represented by the transition DEMCF2 (Demand confirmation). This transition takes a token of the type AV from SELECT and adds it to TOBECF. A token in this place means that the alternative value needs confirmation. Confirmation is the second strategy that is applied when, as a result of the discussion there is no doubt concerning the alternative value to be selected (usually all designers are already committed to one solution). Designers skip the agreement verification procedure described above and go straight to confirm a particular alternative value. This is represented by the transition DEMCF1.

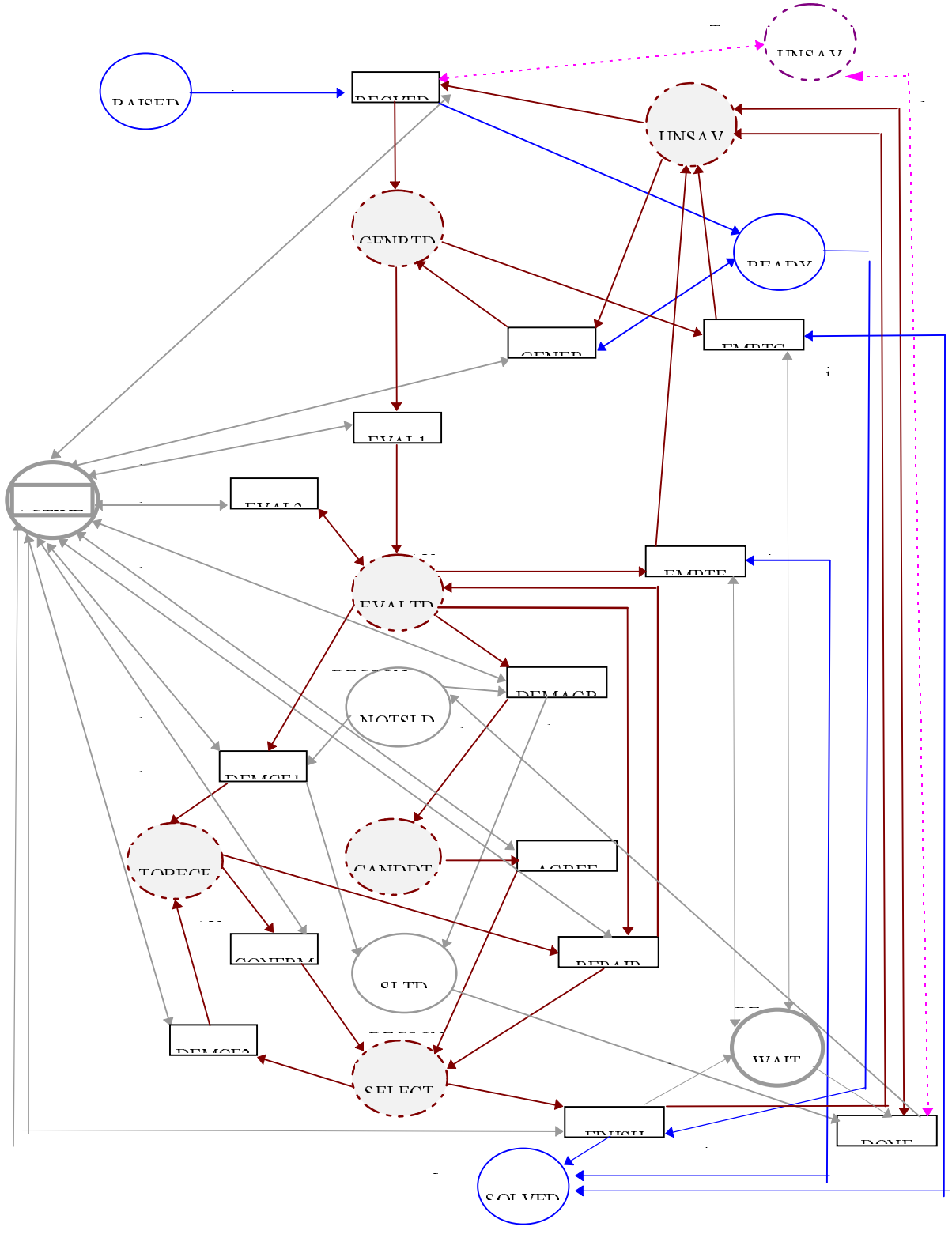

Figure 5. Problem-solving process net

Similarly to DEMAGR (Demand agreement), DEMCF1 also takes a token with value dectkn from NOTSLD and adds it to SLTD to indicate a decision been considered. The selected alternative value is confirmed by restating it, which is represented by the transition CONFRM.

Finally, once a decision was made, the transition FINISH ends the problem-solving process. This transition takes a token of type AV from SELECT and adds a token of the type Issue to SOLVED. This means that the problem-solving process is completed. However, this transition also takes a token of value detokn from ACTIVE and adds it to WAIT.

If a token of value detokn is in WAIT only two transitions are possibly enabled: EMPTE and EMPTG. The transition EMPTE removes tokens of type AV from EVALTD while the transition EMPTG removes tokens of type AV from GENRTD and both add these tokens back to UNSAV. This means that the tokens used to identify alternative values are released, that is, are made available in UNSAV which is a pool of unused alternative value's identifiers. When all tokens are removed from EVALTD and GENRTD, all alternative value's identifiers will be in UNSAV. This condition enables the occurrence of the transition DONE which adds a token of value detokn to ACTIVE, thus enabling the design episode to proceed.

5. Discussion

5.1 HOW TO VALIDATE THE MODEL

In this work we used Coloured Petri nets as a design process description language. It has a formal syntax and semantics which enables precise and unambiguous communication about the design process model. It also provides expressive power, flexibility, and mechanical analyzability.

Our purpose when using a formal operational notation was to validate the proposed model and to analyze its properties. One main analysis aid consists of executing specification (resulting model). Since Coloured Petri nets provide an operational model, it is straightforward to execute specifications by providing an initial marking and then observing one or more possible behaviors or generating the reachability graph. The reachability graph provides all possible behaviors. By observing possible behaviors, one may realize whether or not the model adequately captures the required functionality.

The model can also be analyzed in terms of general properties of nets which establish desirable or undesirable behavior of the model. We used a tool, called ANARCO, to verify three basic net properties, boundedness, liveness and re-initiation, as well as to generate the corresponding reachability graph. ANARCO was developed by the Electrical Engineering Department of the Catholic University of Rio de Janeiro.

5.3 COOPERATIVE DESIGN PROPERTIES

In this section we discuss the testing of the model. Given the reachability graph which provides all possible behaviors, we may test whether or not the model captures the required functionality. These requirements can be expressed in terms of questions and these questions can be verified against the model. This is equivalent to predict the behavior of the modeled phenomenon. It is this prediction capability of the process model that will later be used in a process-centered environment to guide users (designers) in the cooperative design task. Now, let us illustrate these ideas with some possible questions:

Question: If a new issue was already raised is it possible, in the same design episode, to raise a former issue?

If we check all possible behaviors in the reachability graph we will verify that there is not any occurrence sequence in which RAISFI follows RAISNI, that is, RAISFI is never enabled in a marking reachable from a marking resulting from the occurrence of the transition RAISNI. This answer meets the specification since raising a new issue and raising a former issue are mutually exclusive events. The design team must leave the current design episode and then, start a new one in which the former issue may be raised.

Question: If a new issue was already raised, is it possible to raise a nested issue?

If we look for in the reachability graph we will find the occurrence sequence:

Source Marking	Transition	Binding element	Next Marking
0	RAISNI	detokn	1
1	MAKEGO	detokn	4
4	ATTEND	SECDOR	1
1	NEST	(I1,I2)	5
5	SOLVEN	(I1,I2)	18
18	RESUME	(I1,I2)	44

which proves the positive answer to this question.

In the next section we used a formal analysis method - the construction of reachability graphs - to manually verify cooperative design properties. This sort of simulation can be very useful because it gives us a better understanding of the model and can reveal errors (in a similar way to the debugging of a program). Nevertheless, it is not sufficient to give a proof of correctness of the model. Hence, in the following section we use a formal analysis method, the verification of structural properties of CP-nets to characterize the behavior of the model.

5.3 NET PROPERTIES

In this section we present and comment on the results of the analysis of the proposed model performed by a tool, called ANARCO, which considers three basic net properties: boundedness, liveness and re-initiation.

Boundedness is related to the number of tokens in a particular place. Let a place $p \in P$ (set of places). We say that n is an integer bound for p if and only if for every marking reachable from the initial marking the number of tokens for p is less or equal n. In CP-nets we can also define boundedness in terms of multi-set bound. A multi-set m is a multi-set bound for p if and only if for every marking reachable from the initial marking the multi-set for p is less or equal m. This property is important because an unbound net represents a computational system that cannot be implemented.

Liveness is related to the set of binding elements X that remains active. This means that it is possible, for each reachable marking M', to find an occurrence sequence starting in M' and containing an element from X. In other words, this means that all markings reachable from the initial marking lead to the occurrence of some transition. This is important because, otherwise, the net would represent a computational system that could reach a blocking.

Finally, re-initiation is related to the initial marking. A net re-initiates, given an initial marking Mo, if for every marking reachable from this initial marking, there is an occurrence sequence that reaches the initial marking.

We used ANARCO to simulate and analyze the proposed model. Besides allowing the analysis of the model in relation to the properties of boundedness, liveness and re-initiation, the tool also reports flaws in the model, such as, for example, a transition that does not occur.

The group interaction protocol reached 47 markings for the initial marking:

M_0 (IDLE) = 1*SD + 1*EE + 1*DE + 1*GR

M_0 (AVAILB) = 1*t

while the design problem-solving process model reached 333 marking for the initial marking:

M_0 (UNUSED) = 1*NEWI + 1*FORI + 1*QRY

M_0 (UNSDQ) = 1*PRIMQR + 1*SECDQR

M_0 (UNSAV) = 1*AV1 + 1*AV2

M_0 (INACTV) = 1*detokn

M_0 (NOTSLD) = 1*dectkn

and the properties of boundedness, liveness and re-initiation were verified for both. Actually, the two models are components of the cooperative design model. To merge the two models one just needs to add one incoming (and one outgoing) arc from (and to) the place SPEAK - in the group interaction protocol - to (and from) every transition in the design problem-solving process model. Intuitively, this represents that it is needed an speaking designer as the agent of all possible communication actions described in the design problem-solving process model. This alteration does not change the correctness of the model just the performance. Hence, we can interpret the verified properties in relation to the whole model - cooperative design process model - instead of in relation to its components.

The model being bound can be interpreted as representing a constrained process since the number of queries, alternative values or whatever other entity generated in design episode or in a design meeting is limited by real life constraints, such as the duration of the meeting or decision costs.

The model being live guarantees that the net, and consequently the modeled process, is free of fatal blocking and that all parts (regions) in the model are accessible since they represent design problem-solving patterns.

Finally, a model being able to reach its initial marking represents that the design team can always converge to an end and start a new design episode no matter if in the previous one the team reached a solution (decision) or not (and, eventually, the solving of the issue under consideration was postponed to another opportunity in future).

6. Conclusions and Research Directions

In this paper we have proposed a cooperative software design model centered on the design process. We have used Coloured Petri nets as a design process description language. The use of Coloured Petri nets allowed us to generate a comprehensive, concise and unambiguous representation of the communicative behavior of a team involved in a design activity avoiding some disadvantages of other state-based description languages, such as not providing an explicit description of both states and actions.

One of the main advantages of using Coloured Petri nets was that it met our need of identifying and distinguishing particular communication actions and the agents of these actions in order to provide a more accurate model of the activity.

Coloured Petri nets also provide a true concurrence semantics, which means that it is possible to express two or more actions in the same step. In other description languages the actions need to occur after each other, although in any order. This is important because it allows us to describe designers' communicative behavior in a natural, elegant and straightforward way.

The usage of a formal operational notation allowed us to validate the proposed model and to analyze its properties. The existence of computer tools supporting simulation and formal analysis greatly facilitated this task.

Finally, we would like to comment on our research directions. We have developed a formal model of the cooperative software design process. We now intend to use it for the specification of an environment to support this activity. This environment shall be based on the formal model and also on a tool for assisting the design development and documentation of a single designer, named ADD (Garcia 1992). As ADD was originally developed to assist a single designer, our next step is to propose changes required to apply ADD to cooperative design.

7. References

Bandinelli, S. and Fuggetta, A. (1993) "Computational reflection in software process modeling: the SLANG approach." In *Procs of 15th Intl. Conf. on Software Engineering - ICSE 15*. (Baltimore, Maryland).

Bandinelli, S. Fuggetta, A. Ghezzi, C. and Lavazza, L. (1994) "SPADE: An Environment for Software Process Analysis, Design, and Enactment." In. *Software Process Modelling and Technology* (ed.) by A. Finkelstein, J. Kramer, and B. Nuseibech, Research Studies Press Ltd., Taunton, Somerset, England, 223-247.

Conradi, R. Fernström, C. and Fuggetta, A. (1993) "A conceptual framework for evolving software processes." *ACM Software Eng. Not.*

Curtis, B. (1992) "Insights from empirical studies of the software design process." *Future Generation Comp. Syst.,* 7,2/3, 139-149.

Deiters, W. and Gruhn, V. (1990) "Managing Software Processes in the Environment MELMAC." In *Procs of the fourth ACM SIGSOFT Symposium on Software Development Environments*, Irvine, CA, USA, December.

Deiters, W. Gruhn, V. and Schäfer, W. (1989) "Systematic development of software process models." In *Procs of the Second European Software Engineering Conference - ESEC 89*, Springer-Verlag

Fuggetta, A. and Ghezzi, C. (1994) "State of the Art and Open Issues in Process-Centered Software Engineering Environments." *J. Systems Software,* 26

Finkelstein, A. Kramer, J. and Nuseibech, B.(1994) In *Software Process Modelling and Technology*, Research Studies Press Ltd., Taunton, Somerset, England.

Furuta, R. and Stotts, P. D. (1994) "Interpreted Collaboration Protocols and their use in Groupware Prototyping." *In Procs of ACM 1994 Conf on Computer Supported Cooperative Work*, ACM, New York, October 22-24, 121-131.

Gruhn, V. and Jegelka, R. (1992) "An evaluation of FUNSOFT Nets." In *Procs of the Second Workshop on Software Process Technology*, Springer Verlag, September.

Garcia, A.C.B. (1992) "Active Design Documents: A New Approach for Supporting Documentation in Preliminary Design." PhD Thesis, Department of Civil Engineering, Stanford University.

Genrich, H.J. (1986). "Predicate/Transition Nets." in *Lecture Notes on Computer Science 254*, Springer Verlag.

Ghezzi, C. Mandrioli, D Monasca, S. and Pezzè, M. (1991). "A Unified High-Level Petri Net Formalism for Time-critical Systems." *IEEE Transactions on Software Engineering*, 17,2, 160-172.

Gruhn, V. (1992) "Software Processes are Social Processes." Technical Report (2) University of Dortmund.

Harel, D. (1988) "On Visual Formalisms." *Communications of the ACM*, 31,5, 514-530.

Holt, A.W. (1988) "Diplans: A new language for the study and implementation of coordination." *ACM Transactions on Office Information Systems* 6,2, 109-125.

Krämer, B. and Luqi (1991) "Towards Formal Models of Software Engineering Processes." *J. Systems Software*, 15, 63-74.

Maia, A.C.P. Garcia, A.C.B and Lucena, J.C.P. (1995). "A Method for Analysing Team Design Activity." In *Procs of the Symposium on Designing Interactive Systems*, (Ann Arbor, MI, USA), ACM, New York, August 23-25,149-156.

Maia, A.C.P. Fuks, H. and Lucena, J.C.P. (1995) "A model of the communicative behavior of designers involved in cooperative software design" Technical Report PUC-RioInf.MCC 41/95 December.

Snowdon, R.A. and Warboys, B.C. (1994) "An Introduction to process-Centred Environments." In. *Software Process Modelling and Technology* (ed.) by A. Finkelstein, J. Kramer and B. Nuseibech, Research Studies Press Ltd., Taunton, Somerset, England, 1-31.

van Biljon, W.R. (1988) "Extending Petri nets for specifying man-machine dialogues." *Intl. Journal of Man-Machine Studies*, 28, 437-455.

APPENDIX A : Dynamics of Interaction Language

Table 1 presents the group of target communication actions. The occurrence of any of these actions determines the beginning of a design episode.

Table 1. Target communication actions

Table we presents the group of communication actions in which speech objects are supplied.

Table . Supplying speech objects

Table 3 presents the group of communication actions in which speech objects are demanded.

Table 3. Demanding speech objects

APPENDIX B: Other Properties

In [MFL95] we used a state-based notation. State-based notations specify states and transitions between states. In some applications, states represent sets of operational and internal conditions of physical machines, while in others they may represent sets of buttons enabled or disabled, and information displayed or not on the screen of a user interface, and so on. In the applied modeling approach states represent sets of predicates that hold or not as a result of the communicative behavior among designers. Transitions between states are triggered by communication actions. In a particular state a number of communication actions are made available or not for designers use. Hence , the following propeties refer to predicates (in italics) and communication actions (in bold) to state temporal temporal relations between speech objects (representations of verbal communication action) and the conditions for and constraints to enabling the utterance of specific speech objects:

III. There is some design-episode in which during some interaction cycle *nested-issue* is satisfied and **[i]** (to raise a nested issue) is available.

IV. In all design episodes if the event **resume** is available then the *nested-issue* is satisfied.

V. In all design episodes if the event **resume** is available then **discard** is also available .

VIa. There is no design episode in which during some interaction cycle both events **stop** and **resume** are available .

VIb. There is no design episode in which during some interaction cycle both events **stop** and **discard** are available.

VII. There are some design episodes in which the pair of events (**resume** and **discard**) is never available.

VIII. At any interaction cycle of every design episode, or the event **Stop** or the pair of events (**resume** and **discard**) is available.

IX. It is possible to begin a new design episode at any moment.

X. From the beginning of a design episode, it is true that one can never reach an interaction cycle in which no communication action can be taken.

XI. There are some design episodes in which during some interaction cycle *query-posed* is satisfied and **i** and **<i** are available.

XII. At any interaction cycle of every design episode, **q** is available.

XIII. In all design episodes if **[i]** is available then *issue-raised* is satisfied or *issue-revisited* is satisfied.

XIV. There is no design episode in which during some interaction cycle *issue-raised* is satisfied and **i** is available.

XV. There is no design episode in which during some interaction cycle *issue-revisited* is satisfied and **<i** is available.

XVI. There is no design episode in which during some interaction cycle *issue-raised* is satisfied and **<i** is available.

XVII. There is no design episode in which during some interaction cycle *issue-revisited* is satisfied and **i** is available.

XIX. There are some design episodes in which during some interaction cycle *nested-issue* is satisfied and the communication action **Sav** (Supply an alternative value) is available.

XX. In all design episodes if *issue-revisited* is satisfied then *alt-of-value-supplied* is satisfied.

XXI. In all design episodes if the communication action **Sf** (Supply belief) is available then *alt-of-value-supplied* is satisfied.

XXII. In all design episodes, if the communication actions **Da** (Demand agreement) and **Dc** (Demand confirmation) are available, then *alt-of-value-evaluated* is satisfied.

XXIII. In all design episodes, if *alt-of-value-selected* is satisfied, then *selection-candidate* is satisfied or *demanded-confirmation* is satisfied.