ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

C.R. Roast and J.I. Siddiqi, Sheffield Hallam University, UK (Eds)

BCS-FACS Workshop on Formal Aspects of the Human Computer Interface

Proceedings of the BCS-FACS Workshop on Formal Aspects of the Human Computer Interface, Sheffield Hallam University, 10-12 September 1996

A Design Life-Cycle for the Formal Design of User Interfaces

P. Palanque and R. Bastide

Published in Collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

ISBN 3-540-76105-5

A Design Life-Cycle for the Formal Design of User Interfaces

Philippe Palanque^{1,2} & Rémi Bastide¹

¹L.I.S.-I.H.M., Université Toulouse I
Place Anatole France
31042 Toulouse, FRANCE
Tel: +33-61-63-35-88
E-mail: {bastide, palanque}@cict.fr

²Centre d'Etudes de la Navigation Aérienne
7, Avenue Edouard Belin
31055 Toulouse cedex, FRANCE
Tel +33 62 25 95 42
Email: palanque@dgac.fr

Abstract

This paper aims at clarifying the articulation between the task models and system models encountered in CHI design practices. We demonstrate how the use of a formal task model may enhance the design of interactive systems, by providing quantitative results on which designers may base their decisions. We also demonstrate that it is possible to describe both task and system models within the same formal framework. This enables us firstly to formally prove that task and system models comply with each other, and secondly to perform quantitative analysis on the combination of task and system models. We propose a software design life cycle integrating formal task and system modelling. The approach is illustrated by a toy example which, despite its small size, allows us to develop both task and device models and to perform several iterations on the life cycle. The device and tasks are modelled using the Interactive Cooperative Objects (ICO) formalism, which is based on Petri nets and on the object-oriented approach. The formality of Petri nets allows for axiomatic validation of isolated and interacting subsystems.

1 Introduction

Although the use of task models is gaining acceptance in the design of interactive applications [11,14], it is far from being a widespread practice. A reason for this might be the lack of agreement on how task models interact or relate with the models used in the design of the system itself (e.g. data models, object models or dynamic models). The extent to which task models may help in designing models of the behaviour of the system itself is also unclear. This discrepancy between “ anthropocentric - (people) oriented requirements specifications and computer-centric design specification ” [6] has been dealt with in a series of communications in the journal “ Interacting with Computers ”[2,3,7].

We advocate a design practice where task and system models are tightly integrated, and produced together in an iterative process. The design starts either with some initial model of the system (which may originate from the existing situation, e.g. analysis of the paper documents in an information system analysis) or with an initial task model which provides the end-user requirements. Task models and system models are then evolved incrementally. At each iteration of the life cycle, the complexity of task models is quantitatively assessed. The system designers propose modifications in the system models, in order to allow building simpler and more efficient task models. Task models are built in accordance with the new system, and analyzed once again. This iteration will continue until satisfactory task models are produced.

Our approach can be related to that of [16], where the logical notation of Z is used to represent both people and computer systems. By comparison to Z, our approach benefits from the executability of Petri nets and their diagrammatic representation, and puts the emphasis on the dynamic behaviour of humans and systems rather than on their logical description. Petri nets, for example, allow for a compact and readable description of concurrency and synchronisation, which is lacking in other formal approaches such as Z.

Our proposal is significantly different from others such as [24], where models of the system are deduced or generated from task models. In our approach task and system models are not deduced from one another, but on the contrary are evolved in synergy and are checked for compliance with each other.

2 Design Life-Cycle

The position of task modelling within the design life cycle deserves a close examination. Besides, as we advocate the use of formal notations, the position of formal modelling and analysis in this context has also to be made clear.

The advantage of using formal notations to support the design of the models is the potential for mathematical verification they provide. A formal model (whether it describes a task or a system) may be automatically checked for several important properties such as deadlock freedom or termination.

Figure 1 : Design life cycle using formal specification techniques

If, at a given stage, the model fails to comply with such properties, it has to be reworked and corrected until the problem is solved. This process is illustrated in Figure 1.

It is important to notice that such a process may, at the very best, ensure the system under design is “defect free”¹. This process alone can by no means ensure that the system will be usable at all, and much less that it will be “user friendly”.

The solution we propose to this end is threefold :

1. We extend the scope of our formal notation to encompass task modelling. This brings to task modelling the advantages of formal approaches, the most important of which are conciseness, consistency and lack of ambiguity. This also makes task models amenable to mathematical verification.
2. The fact that both task and system model are constructed within the same formal framework enables us to consistently merge task and system models. This in turn allows us to check that task and system models comply with each other (the precise signification of that is detailed later on), and moreover enables us to perform quantitative analysis on the task/system merger in order to check whether the models comply with pre-planned objectives in terms of complexity and timing.
3. We propose a design life cycle supporting the use of formal notations both for task and system modelling. This life cycle (Figure 2) instantiates the life-cycle described in Figure 1 as a sub-cycle to support both task and system formal modelling and extends it to include task/system cross verification.

The life cycle proposed in Figure 2 may start either by some rough model of the system (which may originate from the existing situation, e.g. analysis of the paper documents in an information system analysis) or with an initial task model. The initial model is then submitted to the formal sub-life-cycle of Figure 1. An higher level design loop is then initiated, where task models are built in accordance with the current system model at each iteration, and the complexity of these models is quantitatively analysed. The system designers propose modifications in the system models, in order to allow simpler and more efficient task models to be built. Task models are built in accordance to the new system, and analysed once again. This iteration will continue until satisfactory task models are produced.

This design loop may be undertaken successfully only if the analysis of the task models yields precise and quantitative results, that may be checked against pre-planned objectives. Our approach to this end is to use a sound and well established formal notation to support both task and system modelling. We have chosen high-level Petri nets [10] because of their mathematical foundation, and of the huge amount of work devoted to their formal analysis. We use an object structured dialect of Petri nets (namely the ICO formalism [18]) because the object oriented features of this

¹The extent to which current formal methods may satisfy such a goal is discussed in [12] and [4].

formalism (classification, encapsulation, inheritance) allows us to cleanly cope with the complexity of modern interactive systems. We have devised a set of metrics that can be applied to Petri net models. Those metrics are based on well established Petri net analysis techniques (e.g. the checking of place and transition invariant), but also on weightings applied to the various components of the net (places, transitions, arcs, inscriptions, functions), that will allow us to quantitatively assess the complexity of the analysed tasks.

3 The Formal Method

A lot of work has been more or less recently done in the field of formal approaches for interactive systems [13,15] but really little of it has been directed towards the use of formal methods for the actual construction of interactive systems. Among these our work is very similar to the one in [24] but mainly differs in the design life cycle approach (they consider the design of the system as starting from the task model with no cycles) and on the formalism used (they use LOTOS and a dialect of temporal logic called ACTL instead of Petri nets and objects). More precisely, our formal approach is based on the use of the ICO formalism [18]. This formalism uses concepts borrowed from the object-oriented approach (classification, encapsulation, inheritance, client/server relationship) to describe the structural or static aspects of systems, and uses high-level Petri nets to describe their dynamic or behavioural aspects.

ICOs were originally devised for the modelling and implementation of event-driven interfaces. An ICO model of a system is made up of several communicating objects, where the behaviour and communication protocol of the objects are described by Petri nets.

When we extend the use of the formalism to encompass task modelling as well, the task models themselves are encapsulated by object classes, thus providing the task description with a clearly defined state and behaviour description. The “task object” communicates with system objects in the same syntactical way as system objects communicate with one another [22].

Figure 2: The design life-cycle of an interactive system using both formal task and system models

When two objects communicate, one is in the position of a “client”, requesting the execution of a service and waiting for a result, while the other is in the position of a “server” whose role is to execute the service. In our modelling approach, task objects (which model the behaviour of a user interacting with the system) will most often be in the position of clients (thus modelling the fact that the user applies for a service offered by the system), whereas the system objects will most often be in the position of servers (because in modern interfaces where the software is event-driven, the system is usually passive, waiting for the user to trigger an action).

The fact that the communication protocol as well as the objects’ inner behaviour are both modelled in terms of Petri nets allows us to have a precise semantics for the behaviour of a system modelled by a set of communicating objects. We will make use of this feature in the section “Syntactic Consistency”.

4 A Simple Case Study

To demonstrate our approach, we will present a very simple case study, where the system to be designed is hardware rather than software. However, the kind of problems studied in this case study will be encountered in software design if time-driven evolution is present. Our current work addresses scalability issues, in order to make this approach viable for real life software systems, but those concerns are not presented here for space reasons and can be found in [17].

This case study aims at exemplifying the use of the design life-cycle introduced in the section “Design Life-Cycle”. We first start from an initial model of the system and then show two successive modifications of the system and task models. This corresponds to performing two iterations in the design life-cycle presented in Figure 2. The internal loop for the building of each model as well as the quantitative analysis box of Figure 2 are discussed in the section “Formal Verification of Models”.

4.1 The Initial System

The system available initially consists of a light bulb controlled by a timer. The timer is triggered by a button, and its period is 30 seconds. If the button is depressed while the light is on, the timer is reset and starts over counting down for 30 seconds.

Figure 3 The initial system

Note that this presentation of the system is informal but that this information is absolutely necessary to start building a reasonably detailed model of the task. This information is the minimum that has to be provided to the task analyst in order to build a meaningful task model. If it is not provided, the task model designer will have to make assumptions about the behaviour of the system or else he/she will only be able to discuss about general goals, at a level that will not enhance further our ability to improve the system.

4.2 The System Model

The proposed system is straightforward to describe in terms of Petri nets, as shown in Figure 4. However, let's take some time in describing the Figure, in order to demonstrate various features of the ICO formalism.

Petri nets allow us to describe in a formal and concise way the dynamic behaviour of systems. The state of a system is described by tokens (represented by black dots) distributed in places (graphically represented by ellipses). The tokens' distribution can evolve according to paths made up of transitions (graphically represented by rectangles) and arcs.

Figure 4 The model of the initial system

The timer controlled light offers only one affordance, a button allowing the user to turn it on. The model of this device also highlights other primitives of our modelling approach: In the initial state of the system the light is off which is modelled by a token (a black dot) in the place *LightOff*. From that initial state the transition *PressSwitch* can be triggered by the user, removing the token from the input place *LightOff* and setting it in the output place *LightOn*.

The transition *PressSwitch* depicts a *user service*, meaning that the user has a way to trigger the occurrence of the transition through the associated input device. Graphically such transitions (called *user transitions*) are depicted with input and output unconnected broken arrows, which may eventually bear input or output parameters stating the data provided by the user or returned by the system. The *AutoSwitchOff* transition is an *internal transition*, not related to any

user action, and performed on the system's behalf as soon as it is enabled. The arc between place *LightOn* and transition *AutoSwitchOff* is labelled with the timing inscription [30]. This means that a token staying in place *LightOn* is not available for the firing of transition *AutoSwitchOff* until it has stayed 30 time units (here, seconds) in the place. The effect of the second transition labeled *PressSwitch*, resets the time for the token in place *LightOn* by removing it and putting it back (this is graphically represented by a double arrow). This represents the resetting of the timer each time the switch is pressed. Note that two different transitions are labelled *PressSwitch*, which means that the button affords being pressed in two different states of the system.

4.3 The Task Description

To describe the task, we start by giving the high-level goal it aims to perform, and then detail the steps of the task in terms of actions put at the user's disposal by the system.

In this case, the user's goal is to maintain the light on without interruption for a period of at least 3 minutes. We want to minimise the number of actions performed by the user and thus the solution consisting in pushing and releasing the light switch at very short intervals is not considered relevant¹. In order for the user not to press the light switch more than 7 times the system must be upgraded as it is impossible for the user to perform the desired task without some information about the elapsing of time. In the following sections we will present two possible upgrades of the system, both of them supporting the requested task.

4.4 Upgrade 1: Adding a Stop Watch

Figure 5 The first upgrade of the system

The first possible upgrade is to provide the user with a stop-watch. Thus the system is actually made up of two unconnected subsystems, the stop-watch and the light bulb. In order to use this system, the user needs to use his/her hands to push the light and stop-watch switches and vision in order to watch the elapsed time on the stop- watch. Both the user part and system part are represented in Figure 5.

4.4.1 The System Model

The model of the upgraded system is presented in Figure 6. Now the system is made up of two components, the *TimerControlledLight* and the *Watch*. As those two components are not related, the system model is composed of two unconnected Petri nets. The Petri net corresponding to the *TimerControlledLight* is thus exactly the same as in Figure 4, and another Petri net is added for describing the functioning of the *Watch*. The *Watch* offers two affordances: one button allowing the user to toggle it on and off, and the watch face itself, which allows the current time to be read, but only when the *Watch* is on.

The initial state of the system is described by the initial marking of each Petri net, identical to the one described in Figure 4 plus one single token in the place *WatchOff*. The watch affordances are modelled by transitions in the Petri net, labelled *Press* and *CurrentTime*. The transition *CurrentTime* provides a return value labelled $\langle t \rangle$, the current time when the watch is read.

¹ Special thanks to Alan Dix for calling attention to this possible behaviour of the user

Figure 6 The system model of the first upgrade of the system

4.4.2 The Task Model for the Upgraded System

Given the system defined in the previous section, the task may be described informally as follows :

User presses the light button, and starts the stop-watch (in any order, and within a short time interval)

User measures the elapsing of 25 seconds,

Actions and are then repeated 6 times.

The formal task model corresponding to the above natural language description is described by the high-level Petri net in Figure 7.

Figure 7 demonstrates the surprising complexity of the task associated with this seemingly simple goal. This complexity mainly stems from the fact that the user has to perform a polling loop to consult the stop-watch, and then evaluate the value he/she has just read to decide whether he/she has to press the LightSwitch button once again. This can be seen in the model by the cycle made up of place *ReadyToReadTime*, transition T3, place *CurrentTime* and transition T1.

From the initial state (modelled by one token in both *NeedToStartWatch* and *NeedToStartLight* places and seven tokens in *CountDown* place), the user can press both buttons in any order. After the firing of the corresponding transitions one token is removed from each input place and one token is set in *ReadyLight* and *ReadyWatch*. Then the internal transition *StartCounting* is triggered internally by the user meaning that from that state the user knows that the system is started and it is possible to see the current time on the watch (this is represented in the model by a token set in the place *ReadyToReadTime*).

Figure 7 The task model corresponding to the first upgrade of the system

4.5 Upgrade 2: Adding a Bell

Faced with the complexity in the user's task presented in Figure 7, the designer might choose to improve the system, in order to reduce the user's workload. In this case, a solution might be to move the complexity out of the task and into the system, by ringing a bell 5 seconds before the timer expires.

Figure 8 The second upgrade of the system

The user then only has to count down the number of chimings, which is significantly simpler than to compare several times the current time with the 25 seconds to be awaited before pressing the light switch.

4.5.1 The System Model

The system model is the same as the one of the initial system (see Figure 1) except that an internal transition called *AutoBeep* has been added and that the timing of 30 seconds has been split on two arcs 25s before the ring bells and 5 second before the light switches off.

Figure 9 The System Model

4.5.2 The Upgraded Task Model

With this updated system specification, a new and significantly simpler task model may be built. The count down process is modelled by seven tokens in the *CountDown* place. The complexity of the task model, according to our metrics, is significantly smaller, which proves that the design decision was a good one. Indeed the only cycle of the net is the one corresponding to the repetitive task of pressing the light switch several times. It can be computed that the number of occurrence of the transition *HearBeep* is exactly seven thus stating that the task will correspond to the minimal number of actions requested by the goal.

However, we have not demonstrated that this task is actually supported by the system i.e. that the sequence of user actions embedded in this task model can be performed by the system model given in Figure 9. This is described in the next section.

Figure 10 The upgraded task model

5 Consistency of Task and System Models

This section aims at describing the content of the box entitled *Maintain Task and System Model Consistency* shown in Figure 2, and at showing how it is applied to the case study. When both task and system models have been built their consistency must be ensured at the lexical, syntactic and semantic levels. The use of Petri nets for the design of those models is of great help in order to check this consistency. Indeed, the same mathematical verification that is used for proving the correctness of a model can also be used to prove their mutual consistency.

5.1 Lexical Consistency

This kind of consistency is related to the demand of actions from the task model towards the system model and vice versa. For example if at some point in the task model the user has to press a push-button it is mandatory to ensure that this button exists in the system (and the system model).

This kind of consistency can be automatically proven by checking the action part of the transitions for each model. This means that if there is a service request such as *Light.Pressswitch* in a model (see Figure 10) there exists at least one transition labelled *PressSwitch* in the model of the light, and that the signature of the service and of its invocation match. This resembles the link editing process in programming languages.

5.2 Syntactic Consistency

This kind of consistency is related to the sequencing of actions supplied by the models and their related demand. Indeed, as the lexical consistency corresponds to the existence of a requested action, the syntactic one is related to the actual availability of the action at the moment when it is requested. That is to say that the system can perform any sequence of actions requested by the task model.

The analysis is based on the calculation of the *languages* corresponding to the task and system models. Indeed, a Petri net can be considered as an acceptor for a language in the same way that a finite state automaton can be. The *supply* of a system model is the language defined by its user transitions (see Figure 4) and the *demand* of a task model is the language defined by the transitions where a service request is made of the system (such as transition T3 in Figure 7, which contains the invocation `watch.currentTime()`). Syntactic consistency thus consists of proving that the demand of the task model is included in the supply of the system model. This proof can be performed automatically by building the marking graph of the Petri nets and the application of this kind of analysis to a case study can be found in [20].

5.3 Semantic Consistency

To ensure semantic consistency, some meaning has to be ascribed to the places of the models, and it has to be checked that this meaning is preserved in the interaction between task and system models. The semantic of the places is usually apparent from the name of the places themselves (for example *StopWatchOn*, *StopWatchOff* in Figure 6 of *NeedToStopWatch*, *NeedToStartWatch* in Figure 7). In this case, for example, we have to check that each time the place *NeedToStopWatch* is marked in the task model, the place *WatchOn* is marked in the system model. The formulas to be checked have to be given by the designer (in terms of place invariant) but the checking itself can be carried out by automatic means.

The way to ensure semantic consistency is to merge task and system models. This merging is possible because the protocol for the communication between two Petri nets is itself defined in terms of Petri nets. It is therefore possible to merge the nets pertaining to the task and system models and to perform analysis on the merger. Figure 11 presents a partial reconstruction of the global models made by the combination of the task model in Figure 9 and system model in Figure 10. The transitions related to the operation of the switch (both on the user's and the system's side) have been

expanded and linked. It is important to consider firstly that this global model may be constructed by completely automated means, and secondly that it is not meant to be read by humans, but merely to be processed by automatic tools, in order to perform analysis and prove properties. The result of this analysis on the reconstructed net allows us to be sure that the models have been built in such a way that they can cooperate together without altering their inner behaviour. For example, using Petri net theory it can be automatically proved that the reconstructed net allows the place *CountDown* to be emptied and that there will be no token in place *LightOff* before the place *CountDown* is actually empty, stating that the light will be kept on.

Figure 11 : Task and system models combined

6 Formal Verification of Models

As shown in Figure 1 and in Figure 2 there are two boxes related to verification of models in the design life-cycle: the formal analysis one and the quantitative analysis one. In this section we describe the principles of the verification process. Due to space reasons, the verification is not performed on the case study.

6.1 Formal Analysis

The aim of the formal analysis is to ensure that there are no flaws in the models. The analysis is done by using the mathematical tools provided by the Petri net theory. Using those tools, one can prove general properties about the model (such as absence of deadlock) or semantic domain related properties (i.e. the model cannot describe impossible behaviour such as the Light is on and off at the same time). In [18] we have described general properties that can be proven on a formal model of an interactive system while [19] has shown in a complete example how this kind of analysis can be performed. We give hereafter three properties that have been proven on the system models: the absence of deadlock and the reinitialisability.

1. the **absence of deadlock** is the possibility for the user to issue another system command whatever was the sequence of previous ones;
2. the **reinitialisability** is the possibility for the user to reach the initial state of the system, or a given predefined state (this is very important when the system has to be used several times for the same purpose e.g. an Automated Teller Machine);

Those two properties can be automatically proven on the system models. If those properties were not proven the designer would have to modify the model of the system until they are verified. The verification of deadlock-freedom is done by the calculation of repetitive components in the Petri net (see [19] for more information) which are a set of cycles. The result of the analysis can state for example that a given transition is not in any cycle. Thus the analysis itself can help the designer to improve the design by precisely stating where are the flaws in the models.

The properties to be proven on the task model are different from the ones on the system model. For example absence of deadlock is very important for the system as it is necessary for proving the reinitialisability. At the opposite

the presence of deadlock is very important within the task model as it is necessary for proving the termination of the task which we usually use for proving that the goal of the user has been reached.

It can be easily proven that the task models presented in Figure 7 and in Figure 10 are not live and thus not reinitialisable. Numerous tools are available for doing this kind of analysis such as the ones presented in [8].

6.2 Quantitative Analysis

The use of a formal model for task modelling results in the same advantages as formal system modelling but, as the task model describes the sequences of actions the user will have to perform in order to reach a given goal, it is very important to compute some performance evaluation on the model. This is fully supported using stochastic or timed Petri nets¹ as they have been used for a long time for performance evaluation of systems and a lot of theoretical work is available in this area.

The kind of analysis results that can be done on the task model are:

1. number of actions the user has to perform in order to reach the goal,
2. the number and the length of cycles in the models
3. if some actions have to be performed by the user under temporal constraints (such as entering a password within a given amount of time) it is possible to compute the frequency of those actions and to prove that the temporal constraints are consistent i.e. they do not contradict each other

Another kind of quantitative analysis is related to the complexity of the tasks the user has to perform. This is done by automatically building the marking graph of the Petri net and computing complexity measures on it. For example the number of nodes (corresponding to the number of states in the task model) the number of actions (corresponding to the number of arcs with different labels) and the length of the path to come back to the initial state are associated with weights in order to have a quantity of complexity of the task model. This quantity is used within the design life-cycle to decide whether or not the task and system models meet the requirements.

7 Conclusion and Perspectives

This paper demonstrates the possibility and the benefits of using a common formalism to describe both the system and the tasks of users interacting with it. To this end, we use a formal approach, based on the use of Petri nets and structured according to object-oriented concepts. We also propose a software design life cycle to support the use of this formalism in a sound way.

This work is currently integrated in a software platform in order to make the design and the analysis of the models as automated as possible. This model-based environment [9] integrates an Interactive Cooperative Object editor, an Interactive Cooperative Object interpreter that generates C++ classes (see [1] for more information on the system) and will embed very soon some analysis tools already available in the Petri net community (an overview of the tools available can be found in [8]).

The ongoing work on this approach consists of extending the modelling to encompass user models as well as task and system models. We are currently working on the inclusion of temporal values of the human processor [5] in the task model in order to quantitatively evaluate the cognitive workload of the user. This approach allows us to fully represent the formula $\text{Task} + \text{User} + \text{Computer} = \text{System Performance}$ that can be found in [5] p. 406. Indeed, the user's behaviour is included in the task model which is made consistent with the computer model and the performance evaluation is performed using techniques available in timed Petri nets [23].

8 Acknowledgements

The present work owes a lot to the working sessions held in the LIS laboratory in Toulouse, with the LIS CHI team and part of the PII team in the CENA, Toulouse. The authors are also involved in a related work, together with Tom Moher and Victor Dirda, from the University of Illinois at Chicago.

9 References

1. Bastide R. & Palanque P. (1995) A Petri net based Environment for the Design of Event-Driven Interfaces. Proceedings of the 16th International Conference on Petri nets, Torino, Italy, Lecture Notes in Computer Science n°935, pp. 66-83.

¹ Timed Petri nets is a dialect of basic Petri nets where time is added on transitions, places or arcs of the Petri net, thus allowing us to describe quantitative temporal behaviours in the models (see [21])

- 2.. Benyon D. (1992) The role of task analysis in systems design, *Interacting with Computers*, 4, 1, 102-123.
3. Benyon D. (1992) Task analysis and system design: the discipline of date, *Interacting with Computers*, 4, 2, 246-259.
4. Bowen J.P. & Hinchey M.G. (1994) Seven More Myths of Formal Methods. *Proceedings of Formal Methods Europe (FME'94)*, Springer Verlag.
5. Card S.K., Moran T.P. & Newell A. (1983) *The psychology of Human-Computer Interaction*. Lawrence Elbaum Associates.
6. Diaper, D. (1990) Simulation: a stepping stone between requirements and design in "Simulation and the User Interface", Taylor and Francis, pp 59-72
7. Diaper D. (1992) Task analysis and systems analysis for software development, *Interacting with Computers*, 4, 1, 124-139.
8. Feldbrugge F. & Jensen K. (1987) Petri Net Tool Overview.. *Advances in Petri Nets 1986, Part I & II. LNCS 254 & 255* Springer Verlag pp. 20-61.
9. Foley J.D. & Sukavirya P. (1993) A Second generation User Interface Design Environment: The Model and The Runtime Architecture. *Proc. INTERCHI'93, Amsterdam*, pp. 375-382.
10. Genrich, H.J. (1991) Predicate/Transition Nets, in K. Jensen and G. Rozenberg (Eds.), *High-Level Petri Nets: Theory and Application*. Springer Verlag, Berlin, pp. 3-43.
11. Gray, W.D., John, B.E., and Atwood, M. (1993) Project Ernestine: Validating a GOMS Analysis for Predicting and Explaining Real-World Task Performance. *Human-Computer Interaction* 8(3), pp. 237-309.
12. Hall J.A. (1990) Seven myths of formal methods. *IEEE Software*, 7, (5), pp. 11-19, September.
13. Harrison, M.D. & Thimbelby H.W (eds.) (1990) *Formal methods in Human-Computer Interaction*, Cambridge University Press.
14. Hix D. & Hartson H.R. (1993) *Developing User Interfaces: Ensuring Usability Through Product & Process*. Wiley Professional Computing.
15. Johnson C.W. (1992) A probabilistic logic for the development of safety critical interactive systems. *International Journal of Man-Machine studies*, 38, 2, pp. 333-351.
16. Killip, L.T. (1991) *Fast and Structured Task Analysis* Msc Thesis, University of Liverpool.
17. Palanque Ph., Bastide R., Sibertin C., Dourte L. (1993) Design of User-Driven Interfaces using Petri nets and Objects ; *CAISE'93. Lecture Notes in Computer Science n° 685*, Springer-Verlag.
18. Palanque Ph. & Bastide R. (1994) Formal specification, design and verification of interactive systems using the Interactive Cooperative Object formalism. *Eurographics workshop DSV-IS'94, Carrara, Italy*, Springer Verlag.
19. Palanque Ph. & Bastide R. (1995) Verification of an Interactive Software by Analysis of its Formal Specification. *Proceedings of Interact'95, Norway*, Chapman et Hall.
20. Palanque Ph. & Bastide R. (1995) Formal Specification and Verification of CSCW using the Interactive Cooperative Object Formalisms. *Proceedings of HCI'95, People and Computers X, Huddersfield, U.K.*, Cambridge University Press.
21. Palanque Ph. & Bastide R. (1995) Time modelling in Petri nets for the design of Interactive Systems. *GIST workshop on Time, Glasgow, July 1995*.
22. Palanque Ph., Bastide R. & Sengès V. (1995) Validating Interactive System Design through the Verification of Formal Task Model and System Models. *Proceeding of 6th EHCI'95, Grand Targhee Resort, August*, Chapman et Hall.
23. Palanque Ph. & Bastide R. (1995) Performance evaluation as a tool for the formal design of interactive systems. *Proceeding of CESA'96 IEEE SMC conference, Lille, France, 9-12 July*, IEEE Press.

24. Paterno' F. & Mezzanotte M. (1995) Formal verification of Undesired Behaviours in the CERD case study. Proceeding of 6th EHCI'95, Grand Targhee Resort, August, Chapman et Hall.