Présentation des travaux en vue de l'obtention de

Doctorat de l'Université Toulouse 1

Spécialité ***: INFORMATIQUE***
par

Carmen SANTORO

A Task Model-Based Approach for the Design and Evaluation of Innovative User Interfaces

Soutenue le … devant la commission d'examen composée de :

C. JOHNSON	**Rapporteur**	*Professeur à l'Université de Glasgow, United Kingdom*
G. SZWILLUS	**Rapporteur**	*Professeur à l'Université de Paderborn, Germany*
P. PALANQUE	**Directeur**	*Professeur à l'Université Paul Sabatier, Toulouse*
F. PATERNO'	**Co-Directeur**	*Senior Researcher à l'Institut ISTI-CNR, Pisa, Italie*
R. BASTIDE	**Examinateur**	*Maître de Conférence à l'Université de Toulouse 1*

Laboratoire d'accueil: Institut de Recherche en Informatique de Toulouse

Contents

1. Introduction

The importance of the user interface component of interactive applications is being more and more recognised and a lot of effort is being spent in the academic and industrial community in order to make user interfaces able to support a wide variety of activities, in a wide variety of contexts and available to a larger and larger number of users. Theoretically, this situation should provide users with a high level of flexibility, efficiency and productivity while reducing errors and need for training. Unfortunately, more than ever it reveals to be a source of frustration for users interacting with rather unusable applications. Developing highly usable interactive systems is then becoming one of the chief concerns for many companies and organizations. However, this is far from being a trivial process, since there is a plethora of aspects that might quickly become burdensome for designers striving to build user interfaces able to effectively support the activities users intend to perform.

One of the most well-recognised techniques for the design of interactive systems is the use of models, whose main characteristic is considering and representing some aspects of a problem while leaving out other ones deemed not relevant for a specific objective. This approach is particularly useful when it comes to manage the complexity of medium-large real applications. The purpose of so-called *model-based approaches* is exactly to identify relevant abstractions allowing designers to specify and analyse interactive software applications from a more logical point of view, rather than being forced to immediately address the low level implementation details.

In the human-computer interaction area, the research community has shown interest for many models, ranging from those specifying the activities of users interacting with the application, to other models for describing the underlying system, as well as models for representing mental activities going on in the mind of the user, not to mention models for specifying the particular context in which the different activities take place, and so forth. Such different descriptions can be more or less relevant depending on the different stages of the lifecycle of a system. For instance, during the design phase we could be more interested in the structure of the software, so we need models for specifying the components to be built, their interfaces and relationships, which is quite different from the models that we might use during the requirements analysis phase, when we want to investigate and understand the customers' demands. Moreover, a model could be used in combination with other ones, in order to check consistency between the information contained in the different models.

In spite of such a multifaceted scenario, a general agreement exists about the central role played in the design of interactive applications by *task models*, which describe the activities the users have to accomplish in order to reach their goals. As a matter of fact, almost all model-based approaches include some kind of task models and the reason is that not only they support, due to their own features, a user-centred approach for building interactive systems, but also because they reveal to be extremely ‘versatile’ as they can be useful in various stages of the development of interactive applications, not only the design, but also the verification and evaluation phases.

However, it is worth pointing out that it makes a considerable difference whether a safety critical system or a mass-market consumer product has to be designed. In this thesis we will propose a task model –based approach that was particularly geared to design user interfaces for safety critical applications, which are applications that have to control real-world entities while avoiding that such entities reach hazardous states (states where there is actual or potential danger to people or the environment). There are many examples of safety-critical systems in real life (air traffic control, railway systems, industry control systems, ...). In this field specific issues arise: for instance, sometimes user actions cannot be undone (for example, if an irreversible physical process has been activated), so the issue of *user errors* and how to design

the user interface so as to avoid them, acquires a special importance because accidents often are caused by a human error whose likelihood may be increased by poor design.

In this dissertation we will draw our attention to interactive safety-critical applications, and we will show how the use of task models can support the different phases of the process of building such applications: from design, to verification, to evaluation.

1.1. OBJECTIVES

The central message of the thesis is that a thorough analysis of task models is fundamental for obtaining highly usable interactive systems especially in safety critical contexts where the effects of using interactive applications poorly designed might be catastrophic.

This dissertation will thoroughly describe how to operatively use such models to support the process of building such systems and a notation for specifying task models (ConcurTaskTrees or CTT), together with the related tool (CTTE), will be used in order to illustrate the main features of the approach. The particular suitability and effectiveness of both notation and tool in supporting specific techniques used within the proposed approach will be also highlighted.

The goal of this work is then providing designers with a (task model based) *method* able to effectively cope with the specific requirements that safety critical applications raise. Such a method is articulated in a number of phases in which task models intervene. More specifically, we will describe the benefits that can be afforded by using task models in the requirements elicitation and specification phase, by requiring precise definition of temporal relationships between the different activities that should be performed, so avoiding any ambiguities.

Furthermore, we will describe how task models may be operatively exploited in the design process beyond early analysis as they can provide valuable information for the design of interactive applications through a number of criteria specifying how to use the data contained in task models to drive the design of the user interface in a consistent way.

Additionally, we analyse how they can be used in the verification phase, to check some properties of the modelled system, so improving the level of confidence towards it, which can be relevant especially in safety-critical contexts, like the Air Traffic Control (ATC) domain we consider as case study in this work.

Moreover, task models might be combined with other models in an integrated approach, in order to check consistency of the information contained across the different models. We will address the problem of bridging the gap between task modelling and system modelling by means of scenarios, sequences of tasks that will be mapped onto sequences of actions performed by the system.

Lastly, we show the benefits that can be gained from using task models in the usability evaluation phase, through a systematic analysis of the impact that the deviations from an expected task plan could have on the safety and usability of the overall system. We believe that, instead of relying solely upon empirical testing in later phases of the development lifecycle, it is vital to perform evaluations of different user interface proposals.

1.2. OUTLINE

The dissertation is structured in the following way:

Chapter 2 is dedicated to introduce the reader to some basic concepts and terminology related to the task world, which will be used through the whole dissertation. In addition, it will provide main features of relevant representations proposed in the related field.

Chapter 3 describes in detail the notation for modelling tasks that will be used in this work: the CTT notation, and the associated tool, the CTT Environment (CTTE), both developed in our group.

Chapter 4 defines an approach that has been proposed for the design and evaluation of safety critical interactive application. Within this framework, task models play different roles, which will be detailed in the next chapters.

In Chapter 5 we describe how information contained in task models can drive the design of interactive applications, with the purpose of ensuring that the user interface derived will be designed in a consistent way with respect to the information contained in the model.

Chapter 6 focuses on the importance of performing early evaluation and verification especially when safety critical systems are considered, and proposes an integration between tools for task modelling with tools for performing verification of properties (model checkers).

Chapter 7 describes an integration framework in which the view proposed by task models are combined with the view provided by system models, descriptions more oriented to specify the internal structure and behaviour of the system. The expected goal is checking consistency between the two representations.

Chapter 8 describes a technique for analysing what happens when some deviations occur with respect to the planned behaviour specified within the task model. This technique reveals particularly effective in safety-critical contexts, where user error could threaten human life.

Chapter 9 revisits the concepts illustrated in Chapters 4-8, applying them to a case study in the ATC domain. The purpose of this chapter is basically to show the scalability of the proposed framework to a real application.

2. Task Model -Related Concepts

Summary

In this chapter some basic terminology regarding task –related concepts is treated. The objective is to introduce the reader with key terms that will be used throughout the thesis. We start with an outlook on model-based approaches, then describe the Norman's action model, a framework describing the cognitive process at the basis of any human interaction. Then, we provide the reader with concepts concerning task analysis and modelling, together with some insights on the most relevant techniques already put forward in this field.

2.1. MODEL-BASED APPROACHES

A model can be defined as the representation of some abstraction for a part of reality. Whether it exists in a person's head only, or it exists as an external artefact (for example as a diagram on paper, or a piece of text or mathematics), when we speak about models we usually suppose that some aspects are included in the abstraction while we leave out of the abstraction other ones. This process underlies all modelling techniques and in this respect every model offers a particular 'perspective' of the world that has be represented.

There are several advantages in using models: for example they support highlighting important information and help to manage more easily the complexity of interactive applications during their development, maintenance and continuous improvement. Nevertheless, they have not been exempted from criticisms, one of the most traditional is the that generally the benefits gained using models are not always justified by the cost of extra-time required by developing them.

This issue might be questionable especially in safety-critical systems where it is the human life to be at stake: in these domains the overall cost-benefits ratio of using becomes far more acceptable. The collection and analysis of data can be time consuming, but as with other task analysis methods, the costs of not performing such analyses are high due to the need to correct later errors in systems development.

The purpose of model-based approaches is just to identify relevant models allowing designers to specify and analyse software artefacts from a more semantics-oriented point of view, rather than being forced to address immediately the low level details of the implementation level. During the lifecycle of an interactive system, there are a number of distinct but related models we can consider. For instance, with a *task model* the focus is on the activities users undertake in order to reach their goals, whereas in a *domain model* the objects that a user can access and manipulate through a user interface are defined, whereas a *user model* mainly focuses on the properties of the expected users, tying to categorise them according to some individual user preferences and characteristics. In addition, at different stages, one model might have more relevance than another one. For instance, during the design stage we are interested in the software structure, then we need a *system* model of the components (modules or classes) that we are building, their interfaces and the relationships between them, which is quite different from the models that we might build during analysis, when we would be exploring the nature user requirements. Therefore, one important design choice is to select the most appropriate model for the current goal.

Over the years, a number of model-based approaches have been proposed. UML (Booch *et al.*, 1999) is one of the most successful examples, as it has become a de-facto industrial standard widely accepted in the engineering community. However, among the nine representations provided by UML to support the various phases and parts of the design and development of software applications, none of them is particularly oriented to supporting the design of user interfaces. Of course, it is possible to use some of them to represent aspects related to the user interface (for example activity diagrams can be used to model the tasks considered), but it is clear that this is not their main purpose.

In User Interface Development Environment (UIDE, Foley and Sukavirya, 1994), the authors structure the logical models in terms of a number of user interface interaction objects, with some *pre-conditions* and *post-conditions* associated. Pre-conditions have to be satisfied in order to make the interaction object reactive, whereas post-conditions indicate the effects triggered by a user interaction with the related object, in terms of user interface state modifications.

Humanoid (Szekely *et al.,* 1993) provided a declarative modelling language consisting of five independent parts: the application semantics, the presentation, the behaviour, the dialogue sequencing, and the action side effects. One of its purposes was to overcome the lack of support for exploration of new design that requires undoing portions of the existing design and adding new code. This is one of the limitations of the traditional tools for user interface development, and the reason lay in the fact that such tools do not have a notion of what is specific to a certain design, because this information is tightly embedded in the code.

A task-driven approach can be found in Adept (Wilson *et al.*, 1993), in which three main parts are addressed: the design of a task model, an abstract architectural model, and implementation. The task modeller provides a graphic environment to build and modify task models. The output of this component is used to feed the Abstract Interface Model (AIM) to generate a high level specification of the interaction, expressed in terms of the dialogue structure and abstract interaction objects. A generator tool created a default Concrete Interface Model (CIM) that can be edited by the designer. The CIM can be translated into a platform dependent implementation based on a standard set of widgets.

The MOBI-D (MOdel-Based Interface Designer, Puerta, 1997) approach is a model-based interface development environment for single-user interfaces that enables designers and developers to interactively create user interfaces by designing interface models. The environment integrates model-editing tools, task elicitation tools, an intelligent design assistant and interface building tools.

Almost all model-based approaches include some sort of task models, as they have been agreed as fundamental in user interface design. Most of the reasons stems from the fact that representing the logical activities that should be supported in order to reach user goals allows supporting a user-centred approach not irrelevant when it comes to develop software artefacts like user interfaces.

In the remainder of this chapter we will show some of the main representations that have been proposed up to now in the field of task -related approaches for user interface design. However, before going in further details in the task world, in the next section we deem useful to recall the Norman's action model, a theory at the foundation of the human-computer interaction field, whose main relevance is in the attention paid on better understanding how a user-system interaction works in cognitive terms, thus giving a structured indication of the main aspects to consider for designing an interactive application.

2.2. NORMAN'S ACTION CYCLE MODEL

The Norman's model of human action (Norman, 1988) provides a theoretical framework for the definition of the basic cognitive steps intervening in a user interaction. It is articulated in the following seven stages:

- establishing the goal
- forming the intention
- identifying the action
- executing the action
- perceiving the system state

- interpreting the system state
- evaluating the outcome

Actually, such seven stages are in turn organised in a repeated execution-evaluation cycle (see Figure 1), which starts with users formulating their goals. Such goals could be rather vague, then they need to be translated in more precise actions required to support the related intentions and meet the goal. Then the users will carry out such actions and, after that, they will observe any modification occurring in the system state as a reaction of the execution of their actions. Such changes should be interpreted and the result should be evaluated in order to estimate whether the goal has been reached or not. In the latter case, the goal has to be reformulated and the cycle is repeated.

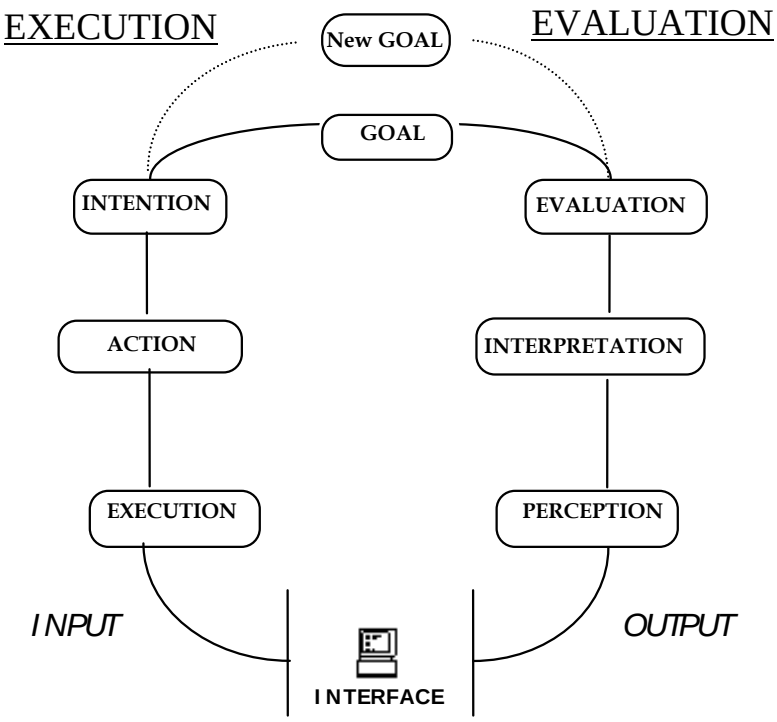

Figure 1. The Norman's Model of Action

In user interface with low usability, where the possible tasks might be badly supported because of mismatches between user's and system's views, two gulfs can be identified: the gulf of *execution* and the gulf of *evaluation*. The gulf of execution involves the difference between the user's intentions in terms of actions to reach and the allowable actions, while the gulf of evaluation represents the difference between the system's representation and the users' expectation. For both the gulfs the notion of cognitive distance (the amount and quality of information processing needed to fill the gap between two states of the considered gulf) can be applied. For action execution, it refers to the information processing needed to bridge the gulf between an intention and the physical actions by which the intention is communicated to the system. In other words, it refers to the act of translating the thoughts and goals of the user into the system's language. For result evaluation, cognitive distance refers to the amount of mental effort needed to translate the information displayed by the system in the terms of the conceptual model adopted by the user.

All in all, in spite of being a rather simplified framework, the Norman's model manages to highlight the significance of understanding users' goals and the consequent importance of analysing and modelling tasks within user interface design.

2.3. TASK ANALYSIS METHODS

2.3.1. Task Analysis

In order to introduce task analysis and related concepts, some basic terminology needs to be defined. We state a *task* as an activity that has to be performed in order to reach a goal. A *goal* is either a desired modification of the state of an application or an attempt to retrieve some information from an application. For instance, accessing the list of the cheapest hotels in a town is an example of task. When complex tasks are considered, it is natural to split them into more refined units, so the concept of *task decomposition* comes into play. The basic units of activity are generally called *actions* or elementary/basic tasks, which can be identified as the various steps required to complete the task. *Objects* are information entities manipulated by tasks. A *role* identifies a set of tasks, and generally this concepts is introduced when it is needed to discriminate between different responsibilities within the system.

We should distinguish between task analysis and task modelling. *Task analysis* is a method for understanding tasks users carry out through interacting with a system. Usually performed during the requirements elicitation phase, task analysis generally involves interviewing end user and observing them in their work place in order to understand their main objectives and duties. Its relevance has been widely recognised in that e.g. it helps in improving the current design of a software artefact, supports the identification of both requirements and potential problems of new design and, additionally, it can be used to design training material, documentation and user manuals. The result of task analysis might be an informal list of tasks relevant for the considered application, and it represents the input for task modelling, which could be considered a more precise refinement and specification of the first one, with the main goal of building a model which describe precisely the relationships among the various tasks identified.

In the following subsections several techniques for supporting task analysis are described. Afterwards, we will make a step further, moving our attention to a number of approaches that have been proposed as more properly task modelling techniques.

Hierarchical Task Analysis (HTA)

Hierarchical Task Analysis (Annett and Duncan, 1967) is one of the oldest task description techniques. The main idea is that complex tasks are broken down into their constituent elements (subtasks). More specifically, HTA is a process of developing a description of tasks in terms of *operations* and *plans*. People perform *operations* to reach a goal (a goal can be viewed as a desired state of the system under control), while *plans* are conditions specifying when each operation should be carried out. The operations can be hierarchically decomposed and for each decomposition, a new plan can be identified.

HTA focuses on what actually happens, rather than on what should happen, and the basic idea is to iteratively develop the HTA models by collecting data from interviews or documents. The representations used by HTA are easily understandable and clear, then, it is a reasonable technique for a developer to explore using it, as it is fairly simple to be applied.

Task Knowledge Structure (TKS)

TKS is a task analysis methodology aimed at capturing some specific knowledge that is used by humans when they perform an activity. This approach assumes that people do have in their mind defined knowledge about tasks, and such knowledge might be represented through a conceptual structured description called TKS.

A Task Knowledge Structure (TKS) has two main parts, a *goal structure* and a *taxonomic structure*. The goal structure represents a person's knowledge about goals and their enabling states, as sub-goals, procedures and how they are sequenced, while the taxonomic structure

represents knowledge about the properties of task objects, their relationships and associated actions.

Another basic assumption of the theory is that the knowledge contained in a TKS can differ in terms of its representativeness and centrality with respect to the task, which basically means that that knowledge modelled within a TKS can vary to some extent in relevance, as there is some knowledge that is 'more important' in determining the success or the failure of the task.

Another concept in TKS is the concept of *role*, which is identified by the set of tasks that a person occupying that role performs. A TKS may be related to other TKSs by a number of different relations, the two most important ones are the *within-role* and *between-role* relations. The within-role relation defines that within a role each task will have a TKS, so a person is supposed to have the knowledge of the tasks that altogether define a particular role. The between-role relation is defined in terms of similarity of tasks across different roles: in this case a common TKS exists comprising the task knowledge common to each role-task instance.

Moreover, it is assumed that knowledge that is judged to be central to the tasks is more likely to be gathered and transferred to similar tasks in different contexts, as this knowledge is tightly bound to the specific task goal.

Use Cases

Use cases are included in UML (Rumbaugh et al. 1997), one of the most relevant modelling languages, widely accepted and used also in industry. They represent widespread practice used in the requirements analysis phase, with the main objective of identifying what should be performed by a system. A use case defines a goal-oriented sequence of interactions between external actors and the system under consideration, which is treated as a "black box": the internal details of the system are not covered, while the interactions with the system are perceived from outside the system itself. Therefore, use cases capture system requirements from the user's perspective.

Use cases are generally structured with one main flow combined with other variations from the main sequence, e.g., alternative paths triggered by options, error conditions, etc. that may also satisfy the goal, as well as sequences that may lead to failure to complete the service because of exceptional behaviour, error handling, etc. A complete set of use cases specifies all the different ways to use the system, and therefore defines all behaviour required of the system, bounding the scope of the system.

Use cases are generally described in natural language, with the various steps written in an easy-to-understand structured narrative using the vocabulary of the domain. This is engaging for users who can easily follow and validate use cases and can be actively involved in the process of defining the requirements thanks to the intuitiveness (and the informality) of the representation they provide. It is generally questioned that use cases are a very informal and imprecise modelling technique, but, actually, they were probably never intended to be anything else.

Scenarios

In order to properly detail tasks a number of different representations might be considered, albeit they all seem to share a common problem: designers often find them difficult to apply. The reason is that it is not easy to know from scratch the structure and the elements of the task model both when analysing an existing application or envisioning a new one.

Scenarios are a well-known technique often used during the initial informal analysis phase. They provide informal descriptions of a specific use in a specific context of application, so a scenario might be viewed as an instance of a use case, representing a single path through it. A careful identification of meaningful scenarios allows designers to obtain a description of most of the activities that should be considered in a task model.

Scenarios are often used for high level evaluation (Carroll 1995). A scenario is a description of an interaction sequence in a particular context. By interpreting them in a cognitive walkthrough (Polson et al. 1992), a heuristic evaluation (Nielsen 1993) session, or by acting them out, certain claims about the design can be tested. Hence, scenarios help designers to focus on specific aspects of the design. During such an evaluation, design knowledge in the form of guidelines, heuristics, ergonomic criteria or design patterns, is used to determine potential usability problems. Walkthrough methods are typically applied by a group of designers and other stakeholders. This evaluation process is still subjective and the designer's expertise is needed to determine the correct application of guidelines, patterns and heuristics. An advantage of these informal techniques is that "contextual" knowledge about the users and their tasks is usually incorporated. In (Carroll 1995) it is stated that scenarios should always include contextual information.

Scenarios can be also developed on the basis of a task model and the user interface specification is then evaluated in that context. During the evaluation session the results can be explicitly compared to the task model. This also makes it easier to determine whether guidelines have been applied correctly. A well-chosen combination of these techniques can be effective in detecting usability problems, especially concerning the main task and goals of the users.

2.3.2. Groupware Task Analysis (GTA)

Groupware Task Analysis (van der Veer *et al.*, 1996) is a method that emphasises on studying groupware systems and their activities rather than studying single users. In this respect, GTA widens the notion of task model by considering three different aspects, each aspect describing the task world from a different viewpoint, but each related to the others:

- *Agents*: "Agents" often indicate people, either individuals or groups, but may also refer to systems. Roles indicate classes of agents to whom certain subsets of tasks are allocated. Organization refers to the relation between agents and roles with respect to task allocation.
- *Work*: The work can be split into different tasks, with different levels of complexity. A distinction is made between (1) the lowest task level that people want to consider in referring to their work, the '*unit task*' (Card et al. 1983); and (2) the atomic level of task delegation that is defined by the tool that is used in performing work, the '*basic task*' (Tauber 1990). Unit tasks are often role-related. Complex tasks may be split up between agents or roles.
- *Situation*: Analysing a task world from the viewpoint of the situation means describing the environment (physical, conceptual, and social) and the objects in such environment. Objects may be physical things, or conceptual ones like messages, gestures, passwords, stories, or signatures. The task environment is the current situation for the performance of a certain task. It includes agents with roles as well as conditions for task performance.

Additional developments in Groupware Task Analysis include an ontology (van Welie *et al.*, 1998a) for the concepts considered, and a tool, Euterpe (van Welie *et al.*, 1998b), to support such concepts.

2.4. TASK MODELLING

The result of task analysis represents the input for task modelling phase, with some modifications that represent a further refinement of concepts and activities that were roughly identified during the task analysis phase.

As it is also shown in (van Welie, 2001) two main types of task model can be specified: a *prescriptive* one vs. a *descriptive* one. The descriptive one illustrates how the user interacts with an existing system, whereas a prescriptive one describes a future system, generally derived from the first one with some modifications added in order to take into account some new requirements (usually driven by technological enhancements). While the first activity has as its main goal describing the 'current' situation, the second one prescribes how the users ought to use the new system, outlining some solutions for coping with new requirements.

It is worth pointing out that, in order to be meaningful, especially in the latter case (task modelling for the envisioned system), the development of the task model should be the result of an interdisciplinary discussion involving the various stakeholders that should be considered (such as designers, developers, end users and domain experts). This could be useful to obtain user interfaces able to easily support the desired activities thus implying that the user task model (how users think that the activities should be performed) and the system task model (how the application assumes that activities are performed) are close to each other.

The literature in task modelling area provides with a number of different representations that have been proposed up to now, mainly differentiating over syntax (textual vs. graphical), level of formality, set of temporal dependencies that express, and possibility of having automatic tools supporting such notations. In this sections we will provide an overview of the main approaches.

2.4.1. Method Analytique de Description des Taches (MAD)

In MAD (Scapin and Pierret-Golbreich, 1989) a task is represented as a hierarchical tree built from sets of task items represented through a graphics-based notation, while the description of task objects is implemented through a class hierarchy (like in an object-oriented fashion). In MAD we have the concept of *constructors*, describing the time dependencies that hold between the different subtasks of a specific task, namely the order in which tasks are supposed to be executed. Some examples of constructors used to model the different temporal relationships that can occur in a task decomposition are: PAR (parallel tasks), SEQ (sequential tasks), ALT (alternative tasks), LOOP (iterative tasks), OP (optional tasks).

An example of MAD task decomposition is shown in Figure 2. This example refers to SimpleDraw, a simple "toy" drawing application developed in-house for experimental purposes. Figure 2 shows the high-level functionalities of SimpleDraw. To use SimpleDraw, the user must first open a file, then work on that file, and finally must close the file.

Moreover, although the notation used in MAD does not have a formal semantics, templates are used to further detail task properties, collecting functional characteristics of the task (e.g.: pre-conditions and post-conditions, initial state and final state, etc.). Other information about non-functional requirements of tasks like duration, frequency, etc. might be specified as natural language descriptions in the task template. MAD is also provided with software tools which allow computer-supported task modelling.

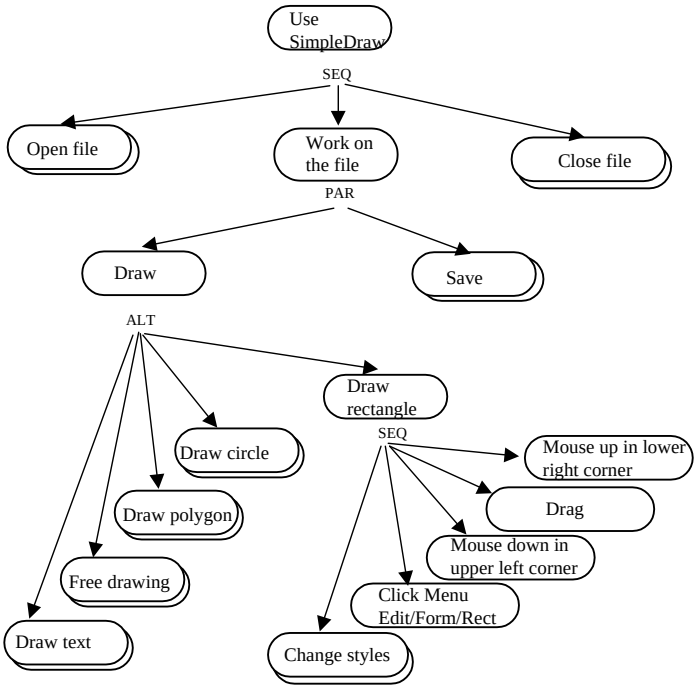

Figure 2: An Example of MAD Description

2.4.2. Goals, Operators, Methods, and Selection Rules (GOMS)

GOMS (Card *et al.,* 1983) is one of the first contributions focusing on identifying the goals that users want to achieve and how to perform activities able to reach such goals. GOMS can be used to model how skilled people will use a system before such a system is actually built, and tries to estimate the user performance based on a description of the system. A GOMS model is composed of *methods* that are possible ways used to achieve specific *goals,* which are states that the user wants to achieve. A method usually contains a number of steps. At the lowest level the methods are composed of *operators*, which describe the action sequences required to reach a goal. If a goal can be achieved by more than one method, then *selection rules* are used to determine the proper method. In Figure 3 an example taken from John & Kieras (1996) is displayed. It models the task of moving text in a Word processor, in the context of editing a manuscript.

```
GOAL: EDIT-MANUSCRIPT
.     GOAL: EDIT-UNIT-TASK ... repeat until no more unit tasks
.     .     GOAL: ACQUIRE UNIT-TASK
.     .      .      GOAL: GET-NEXT-PAGE ... if at end of manuscript page
.     .      .      GOAL: GET-FROM-MANUSCRIPT
.     .     GOAL: EXECUTE-UNIT-TASK ... if a unit task was found
.     .      .      GOAL: MODIFY-TEXT
.     .      .      .   [select: GOAL: MOVE-TEXT* ...if text is to be moved
.     .      .      .       GOAL: DELETE-PHRASE ...if a phrase is to be deleted
.     .      .      .       GOAL: INSERT-WORD] ... if a word is to be inserted
.     .      .      .     VERIFY-EDIT
```

Figure 3: An Example of GOMS Specification

For each operator an average duration might be specified and, by counting all the steps, the total expected time to complete the task is calculated, so, the operators in GOMS are the basis for making predictions about the expected user performance. Although the overall validity of such an estimation remains controversial and questioned by some (as it neglects some important aspects, for example the model does not take into account neither the occurrence of errors nor possible differences among the users), GOMS approach can still be used to compare different design alternatives for identifying the option requiring the minimal number of steps.

There are different versions of GOMS, but basically four are the most important: the Keystroke-Level Model (KLM), CMN-GOMS, NGOMSL, and CPM-GOMS. These four models vary in complexity and are used to model different activities. A detailed discussion of strength and weaknesses of the various versions of the GOMS family can be found in (John and Kieras, 1996).

GOMS-based models are usually just textually represented, which could require some time and effort. In order to overcome such issue, several attempts have been made to develop tools able to support such technique, one example is QGOMS (Beard *et al.*, 1996). However, the tool support provided for GOMS approaches is still far from what designers and evaluators expect from GOMS tools (Baumeister *et al.*, 2000) because of partial coverage of the GOMS concepts, limited support for cognitively plausible models, and lack of integration in the development process.

2.4.3. User Action Notation (UAN)

The main purpose of UAN (Hartson and Gray, 1992) is allowing designers to describe the dynamic behaviour of graphical user interfaces by combining concepts from task models, process-based notations and user actions descriptions. It is a textual notation where the interface is represented through a hierarchical structure of asynchronous tasks, with a number of operators to describe temporal relationships.

A UAN specification is usually structured into two parts: one part describes the task decomposition and the temporal relationships among asynchronous tasks, the other one associates each basic task with information generally stored in a table with three columns: user action, interface feedback and modifications in the state of the interface (see Table 1).

TASK: delete a file		
User Actions	*Interface Feedback*	*Interface State*
~[file] Mv	file!	selected = file
~[x,y]*	outline(file) > ~	
~[trash]	outline(file) > ~, trash!	
M^	erase(file), trash!+	selected = null

Table 1: An Example of Task Description in UAN

Given the rich set of operators available, UAN notation is suitable to specify task behaviours. However, one disadvantage is that it might stimulate large specifications with too many details not always useful for the designer. Another limitation is that the order which has to be followed to interpret the tables of the basic tasks (from left to right) can be rigid and inadequate especially when the modification of the state of the application triggers the performance of the basic task considered. The textual syntax of UAN and its lack of tool support has been a strong limitation to its diffusion.

2.4.4. Activity Diagrams

Although UML was not designed for the purpose of task modelling, there are two representations we deem directly relevant for task –related concepts: use cases and activity diagrams. Instead, activity diagrams involve a greater degree in activities' specification, then they might be considered a technique for task modelling purposes (although not explicitly conceived with these goals in mind).

As their name suggest, activity diagrams focus on activities, and they are generally considered like flow charts supporting compound decisions. However, they differ from flow charts because they explicitly support parallel activities and synchronization through the constructs of fork and join. The biggest disadvantage of activity diagrams is that they do not give many details about how objects behave or collaborate, although labelling of each activity

with the responsible object can be added to diagrams. Activity diagrams might be useful for further analysing a use case by describing what actions need to take place and when they should occur; describing a complicated sequential algorithm, modelling applications with parallel processes, etc.

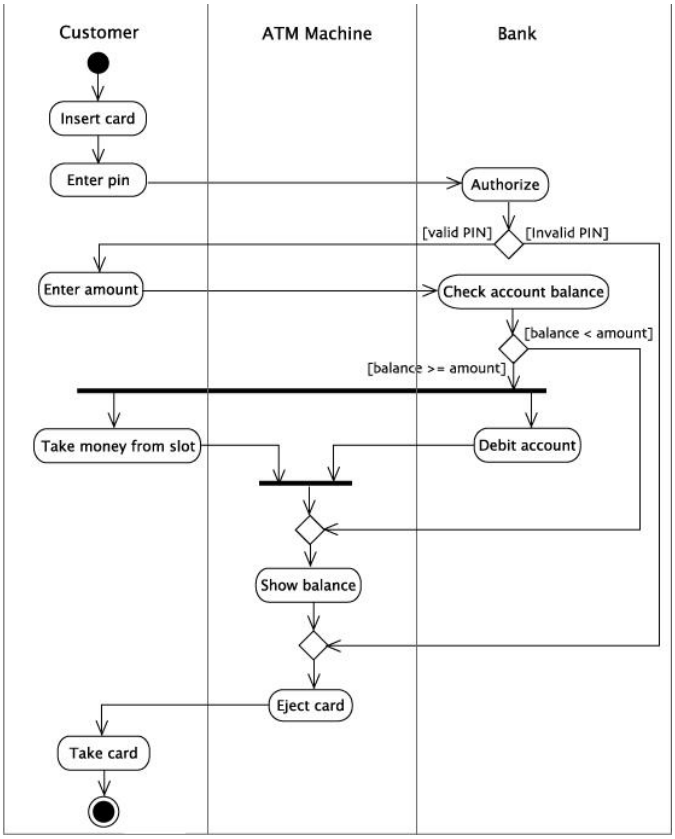

Figure 4: An Example of Activity Diagram with Swimlanes

An activity diagram may be visually divided into *swimlanes*, allowing the designer to specify a set of activities according to who is supposed to perform them. In this way, each swimlane represents responsibility for part of the overall activity, and might correspond to organisational units in a business model. An example of it can be seen in Figure 4.

2.5. CONCLUSIONS

In this chapter a number of different representation for describing tasks have been described. Task models like TKS and GTA can be considered mainly descriptive and focus on modelling the user's task knowledge. Other notations like GOMS are usually prescriptive task models. In the next chapter we will analyse more in depth the features of another notation, the ConcurTaskTrees notation, which is used in a prescriptive way.

3. The CTT Notation and the Related Environment

Summary

In this chapter we present a notation, ConcurTaskTrees, aimed at representing tasks, including their attributes and objects, and their temporal and semantic relationships, to easily create, analyse and modify such descriptions and even simulate their dynamic behaviour. In addition, the main characteristics of an automatic tool –CTTE– that has been developed to support the use of such a notation will be discussed. The tool provides the designer with thorough support for developing and analysing task models of interactive, even cooperative applications.

3.1. INTRODUCTION

In the previous chapter we provided the reader with information on a number of approaches regarding task analysis and modelling. They differ along a number of parameters, which might include the type of representation (textual or graphical), the level of formality, the fact that they might be descriptive or prescriptive and last (but not least) the existence of software artefacts to be delivered to users in order to support their usage. Actually, the existence of tool support might be determinant for the use of a notation, because, as a matter of fact the use of models has often been criticised for the effort required to develop them and the difficulties in using the information they contain to support design and evaluation.

Actually, the first concern after introducing a new notation should be providing users with tools to make its use easier and more effective. The problem is that getting used to another notation involves a significant amount of effort and time spent by the potential users in order to understand features, semantics and meaning of the notation's conventions. Moreover, even when users have understood the main features of the notation, there is still the risk that their effort might be wasted if they find that using it is difficult and not really feasible or appropriate for intensive use and real case studies. One of the strengths of a notation is the possibility of being supported by automatic tools. Developing a formal model can be a long process, which requires a considerable effort, therefore automatic tools can ease such activity and help designers to get information from the models, which is useful for supporting the design cycle.

One of the problems that are usually linked with task modelling is that it is a time-consuming, sometimes discouraging, process. To overcome such a limitation, interest has been increasing in the use of tool support (Bomsdorf and Szwillus, 1999). Some research prototypes was developed years ago to show the feasibility of the development of such tools, however the first prototypes were rather limited in terms of functionality and usable mainly by the people who develop them. More systematically engineered tool-support is strongly required in order to ease the development and analysis of task models and make them acceptable to a large number of designers. If we consider the first generation of tools for task models we can notice that they are mainly research prototypes aiming at supporting the editing of the task model and related information. This has been a useful contribution but further automatic support is required to make the use of task models acceptable to a large number of designers. Thus, there is a strong need for engineered tools able to support task modelling of applications, including cooperative and multi-user ones which are on the rise, and the analysis of such models' contents which may can be complex in the case of many real applications.

While task modelling and task-based design are entering into current practise in the design of interactive software applications, there is still a lack of tools supporting the development and analysis of task models. Such tools should provide developers with ways to represent tasks, including their attributes and objects, and their temporal and semantic relationships, to easily create, analyse and modify such representations and to simulate their dynamic behaviour.

To overcome such problems at HIIS Lab of ISTI/CNR in Pisa, we designed and developed the ConcurTaskTrees Environment (CTTE), a tool that apart from supporting a the CTT notation to express task models, can be also used to support design of interactive

applications. In the remainder of the chapter both of them will be analysed, as they will be extensively referred throughout the dissertation.

3.2. THE CONCURTASKTREES NOTATION

The ConcurTaskTrees notation was developed after first studies developed to specify graphical user interfaces by using LOTOS (Bolognesi and Brinksma, 1987), a concurrent formal notation that seemed a good choice to specify user interfaces because it allows designers to describe both event-driven behaviours and state modifications. However, it showed some limitations that make it unlikely to be widely used in the human-computer interaction domain. It was soon realized that there was a need for new operators to express a richer set of dynamic behaviours in task models in a compact way (such as optional tasks and order independence performance) and additional information useful in analyzing and representing task models capturing their relevant attributes. Moreover, LOTOS has a textual syntax that can easily generate complex expressions, even when the behaviour to describe is quite simple. Thus, a new notation was developed, ConcurTaskTrees. Its main aim is to be an easy-to-use notation that can support the design of real industrial applications, which usually means applications with medium-large dimensions.

The main features of ConcurTaskTrees are:

- *Focus on activities*. It allows designers to concentrate on the activities that users aim to perform that are the most relevant aspects when designing interactive applications, avoiding low-level implementation details that, at the design stage, would only obscure the decisions to take.
- *Hierarchical structure*. A hierarchical structure is something very intuitive; in fact, often, when people have to solve a problem, they tend to decompose it into smaller problems still maintaining the relationships among the various parts of the solution.
- *Graphical syntax*. A graphical syntax often (though not always) is more easy to interpret. In this case, it reflects the logical structure of the notation, therefore it has a tree-like form.
- *Concurrent notation*. A rich set of possible temporal relationships between the tasks can be defined: we want designers to clearly express the logical temporal relationships that should be taken into account in the user interface implementation to allow the user to perform, at any time, the tasks that should be enabled from a semantic point of view.
- *Task allocation*. How the performance of the task is allocated is explicitly represented with four categories: *user* task (only internal cognitive activity such as selecting a strategy to solve a problem), *application* task (only system performance such as generating the results of a query), *interaction* task (user actions such as editing a diagram), and *abstract* tasks (tasks that have subtasks belonging to different categories).
- *Objects*. Once the tasks are identified, it is important to indicate the objects that have to be manipulated to support their performance, which can be either user interface objects or application domain objects. Multiple user interface objects can be associated to a domain objects (for example, temperature can be represented by a bar-chart of a textual value).

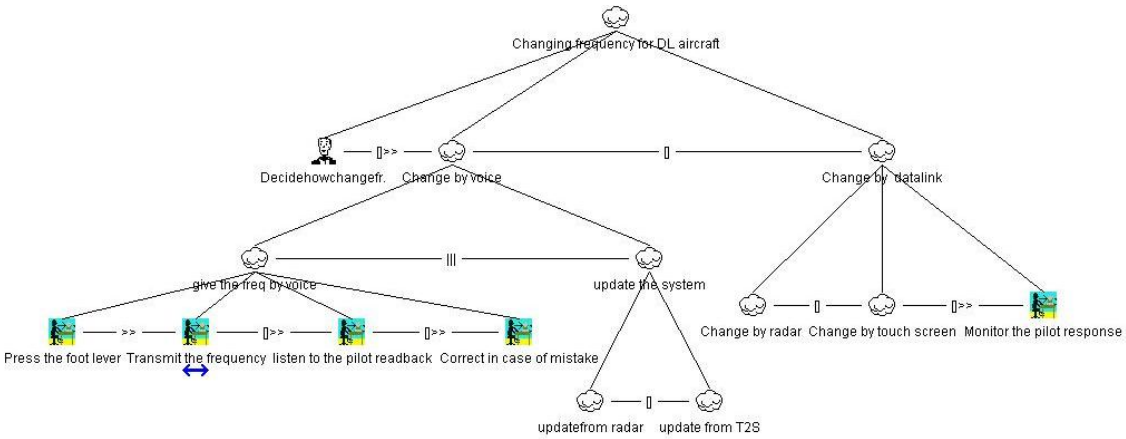

Figure 5. Example of CTT Task Model

The resulting notation allows designers to describe a greater range of behaviours than those that can be described by Hierarchical Task Analysis or GOMS. Richer sets of operators were introduced in UAN, but this notation supports only asynchronous tasks, whereas, we also give the possibility of describing how tasks exchange information. In addition, UAN is a textual notation with no tool support, thus unsuitable for developing and analyzing large specifications.

In Figure 5, there is an excerpt of a task model. We consider the tasks associated to a controller who has to send a different frequency to an aircraft which is data-link equipped (datalink is a particular technology allowing the exchanging of asynchronous messages between controller and pilots). At the beginning the controller has to decide (cognitive task) how to change such a frequency. After that ([]>> is the enabling operator), depending on such a decision, two possibilities are available: either perform this change by voice (see the abstract task *Change by voice*) or ([] is the choice operator) he can perform them through datalink (*Change by datalink*). Such activities are both structured activities, which are in turn articulated in a number of subtasks. For instance, if we focus on the subtask *Change by voice*, it is composed of the activities permitting to properly provide the frequency by voice, concurrently performed (||| is the operator for concurrent performance) with the activity of updating the system. It is worth noting that, as the temporal operators involving parent tasks are inherited by their subtasks, all the elementary tasks composing the task *give the freq by voice* will be concurrently performed with the tasks composing the *update the system* task.

3.2.1. Temporal Operators

A number of temporal relationships have been identified in order to describe how the tasks should be executed. For the moment at least the temporal operators identified are the following ones (they will be listed according to their priority order, from the highest priority to the lowest priority): *choice* ([] operator), *parallel composition* (||| and |[]| operators), *disabling* ([> operator), *suspend-resume* (|> operator), *order independence* (|=|operator), *enabling* (>> and []>> operators). They are operators involving more than one tasks, differently from the *iteration* (T*) and *option* ([T]) that are applied to just one task. Their main meaning is illustrated in the following Table 2.

CTT Operator	Explanation
Independent Concurrency **T1 ||| T2**	Actions belonging to tasks T1 and T2 can be performed in any order without any specific constraints.
Choice **T1 [ ] T2**	It is possible to choose among tasks T1 and T2 but, once the choice has been made, only the chosen task can be performed.
Concurrency with Information Exchange **T1|[]| T2**	T1 and T2 can be concurrently executed and they have to synchronise in order to exchange information.
Order Independence **T1 |=| T2**	Both tasks have to be performed but once one task starts it has to be finished before starting the other one.
Deactivation **T1 [> T2**	The first task T1 is definitively deactivated once the first action of the second task T2 has been performed.
Enabling **T1 >> T2**	A sequential composition exists between T1 and T2: when T1 terminates, it enables the performance of T2.
Enabling with information passing **T1 []>>T2**	Task T1 provides some information to task T2 besides enabling it.
Suspend-resume **T1 |> T2**	T2 can interrupt T1. When T2 terminates, T1 can be reactivated from the state reached before the interruption.
Iteration **T***	Task T is repetitively performed: when it terminates, its performance starts again (from the beginning) until T is deactivated by another task.
Finite Iteration **T(n)**	Finite iteration is used when it is known in advance how many times a task T will be performed.
Optional Task **[T]**	Square brackets indicate that the performance of task T is optional.
Recursion	A high level task T is recursive when one occurrence of it appears within its own specification. No specific operator is associated with recursion.

Table 2: The CTT Operators and Associated Meanings

3.2.2. Single User Tasks

For every task, it is possible to specify a number of attributes and related information classified into three sections (Figure 6 shows how CTTE supports access to each of them):

- *General information*. It includes the identifier and extended name of the task, its *category* and *type*, frequency of use, some informal annotation that the designer may want to store, indication of possible preconditions, and whether it is an iterative, optional, or connection task. While the category of a task indicates the allocation of its performance, the type of a task allows designers to group tasks, depending on their semantics. Each category has its own types of tasks. In the interaction category, examples of task types are *selection* (the task allows the user to select a piece of information), *control* (the task allows the user to trigger a control event that can activate a functionality), and *editing* (the task allows the user to enter a value). This classification is useful to drive the choice of the interaction or presentation techniques more suitable to support the task performance. Frequency of use is additional useful

information because the interaction techniques associated with more frequent tasks need to be better highlighted to obtain an efficient user interface. In addition, as you can see from Figure 6, information about the different platforms that can support the performance of each task is also represented.

- *Objects*. It is possible to indicate the objects that have to be manipulated to perform a task. Objects can be either user interface or domain application objects. It is also possible to indicate the access rights of the user to manipulate the objects. In multi-user applications, different users may have different access rights.
- *Time performance*. It is also possible to indicate estimated time of performance (including a distinction among minimal, maximal, and average time of performance) to allow some performance evaluation of tasks.

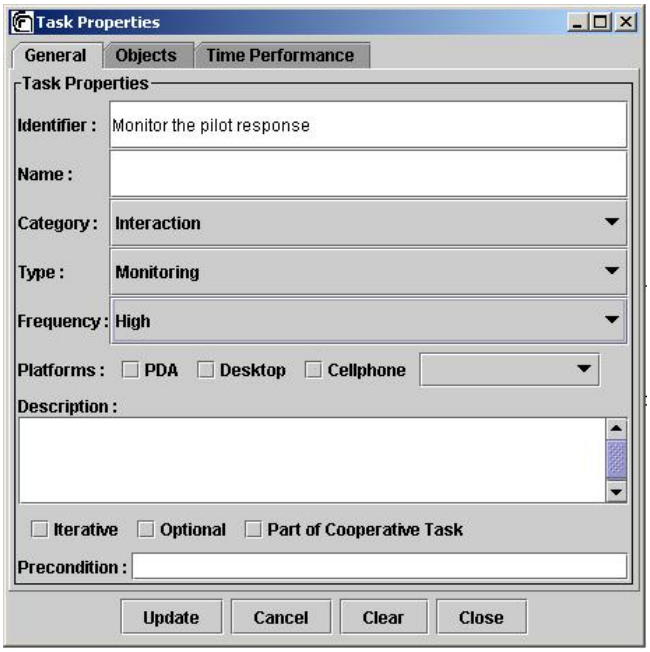

Figure 6. Template to Provide Information Concerning a Task in CTTE

3.2.3. Cooperative Tasks

Providing support for cooperative applications is important because the increasing spread and improvement of Internet connections makes it possible to use many types of cooperative applications. Consequently, tools supporting the design of applications where multiple users can interactively cooperate are more and more required. In our approach, when there are multi-user applications, the task model is composed of various parts. A role is identified by a specific set of tasks and relationships among them. Thus, there is one task model for each role involved. In addition, there is a cooperative part whose purpose is to indicate the relationships among tasks performed by different users.

The cooperative part is described in a manner similar to the single user parts: It is a hierarchical structure with indications of the temporal operators. The main difference is that it includes cooperative tasks: those tasks that imply actions by two or more users in order to be performed. For example, negotiating a price is a cooperative task because it requires actions from both a customer and a salesman. Cooperative tasks are represented by a specific icon with two persons interacting with each other. In the cooperative part, cooperative tasks are decomposed until we reach tasks performed by a single user that are represented with the icons used in

the single user parts. These single user tasks will also appear in the task model of the associated role. They are defined as *connection* tasks between the single-user parts and the cooperative part. In the task specification of a role (for example, in Figure 7 top part the activities carried out by the *pilot* when s/he receives the frequency from the controller are shown), we can identify some connection tasks in that they are annotated by a double arrow under their names.

The effect of this structure of representation is that, in order to understand whether a task is enabled to be performed, we have to check both the constraints in the relative single user part and the constraints in the cooperative part. It may happen that a task without constraints regarding the single user part is not enabled because there is a constraint in the cooperative part indicating that another user must first perform another task.

If we consider both the task model in Figure 5 and the task models in Figure 7, altogether they provide the specifications of activities taking place on the controller's side (Figure 5), on the pilot side (Figure 7, top part) and the collaborations intervening between the different roles (Figure 7, bottom part). If we consider each part of the task model in isolation, it might seem that, for instance, the pilot's task of *Answer wilco* (a sort of confirmation message in atc communications) might be executed just after the completion of both *Hear the DL message notification* and *Visualise the DL message*. However, if we consider the additional constraint indicated in the bottom part of the figure, we can see that the *Answer wilco* has to wait for the additional cooperative constraints indicated in the cooperative tree.

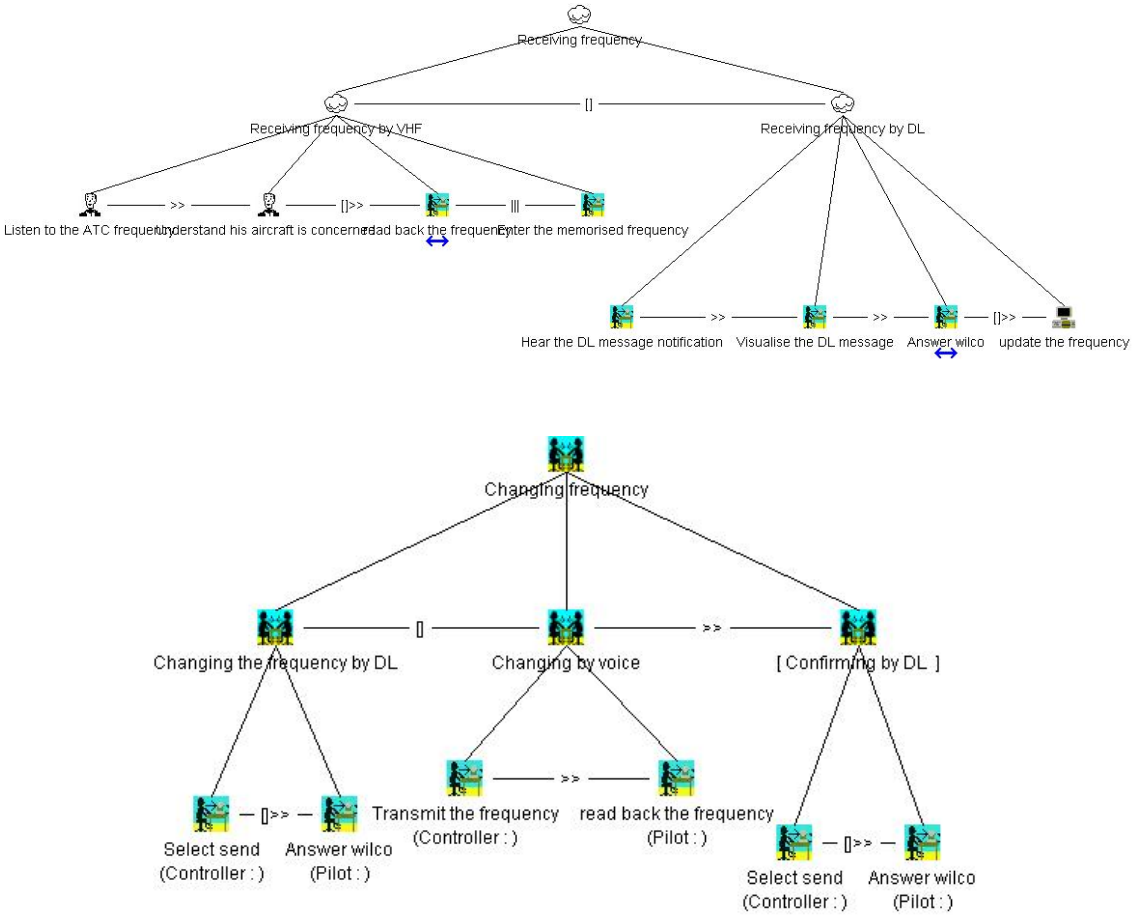

Figure 7. An Example of a Cooperative Task Model (Cooperative Part at the Bottom)

Cooperative work combines communication, individual action, and collaboration. Our task models aim to provide an abstract representation of these aspects. The task model associated with a certain role enables distinguishing both individual tasks and tasks that are related to cooperation with other users. Then, moving on to the cooperative part, it is possible to see what the common goals are and what cooperation among multiple users is required in order

to achieve them. The set of temporal operators also allows describing flexible situations in which the performance of the different activities depends on the occurrence of specific conditions. Of course, specifying flexible behaviours implies increasing the complexity of the specification. Trying to anticipate everything that could go wrong in complex operations would render the specification exceedingly complex and large. The extent to which designers want to increase the complexity of the specification to address these issues depends on many factors, such as the importance for the current project, resources available, and experience in task modelling.

The tool allows browsing the task model of cooperative applications in different ways. The designer can interactively select the frame associated with the part of the task model of interest. In addition, when connection tasks are selected in the task model of a role (they are highlighted by a double arrow below their name), it is then possible to automatically visualize where they appear in the cooperative part and vice versa.

3.3. CTT ENVIRONMENT

In this section we present a tool, CTTE, that provides thorough support for developing and analysing task models of cooperative applications, which can then be used to improve the design and evaluation of interactive software applications. The tool provides a large number of possibilities for supporting easy development and analysis of task models, the richest currently available. It leverages the CTT notation, which allows designers to represent the structure of the task model in a hierarchical way, which has been shown to be an intuitive and modular way (one part for each role plus the cooperative part). In addition, the tool supports different views of the task model (complete view, summary view, view by levels, folding and unfolding of sub trees, etc.). Moreover, the task model's structure can be easily modified with the additional support of an interactive simulator that allows designers to better understand the dynamic behaviour of the model specified. In this section, we describe the features of the tool, how they are supported, and the motivations underlying their inclusion. Section 5 is dedicated to the simulator: its functionality and underlying algorithm.

3.3.1. Editing Task Models

New tasks are added by selecting the category icon; depending on the selection mode, the new task is added either as the last right child or as a left sibling of the current selected task. It is possible to line up a group of tasks, to change the distance between two levels of the tree, to automatically change the layout to avoid overlapping nodes.

The tool allows for the easy copying and pasting of entire sub trees. Nodes, including whole sub trees, can be disconnected and reconnected arbitrarily, so it is easy to restructure the tree. It is possible to construct a library of patterns associated with sub trees for common tasks which the analyst can reuse in future models. It is possible to interactively select a node and expand or collapse the related sub tree. Folding a sub tree can be useful when designers do not want to be disturbed by too many details. The tool is also able to show only a specific number of levels of the task model that is interactively indicated by the user. When the model extends beyond the bounds of the initial window, the tool automatically enables vertical and horizontal scrollbars to allow the designer to still access the entire model. The tool also supports automatic search of a task in the graphical representation of the model.

Moreover, when cooperative task models are developed, the overall model is structured in one model for each role involved and another model for describing the cooperations among users belonging to different roles. There is a tabbed window associated with each part (possible parts are the roles defined and the cooperative part) and, at any time, designers can select which one to bring to the front. To better analyze relationships between cooperative and single user parts, when a single user task in the cooperative part (a connection task) is selected, then the tool automatically shows the task model of the corresponding user and it highlights where it is located within it.

To facilitate the consistent development of the various parts of the task model of a cooperative application when the cooperative part is selected, the designer can select a role from the list on the right side and the tool lists the connection tasks that have been identified for that role (for example, in Figure 8, the roles are Controller and Pilot, Controller has been selected, and, below, the list of related connection tasks is displayed). This allows designers to easily check whether there is any connection task that has not yet been included in the cooperative part and, in that case, they can immediately add it. They just have to create a new task and the name and the role of the connection task will automatically be associated to the new task by selecting the associated connection task.

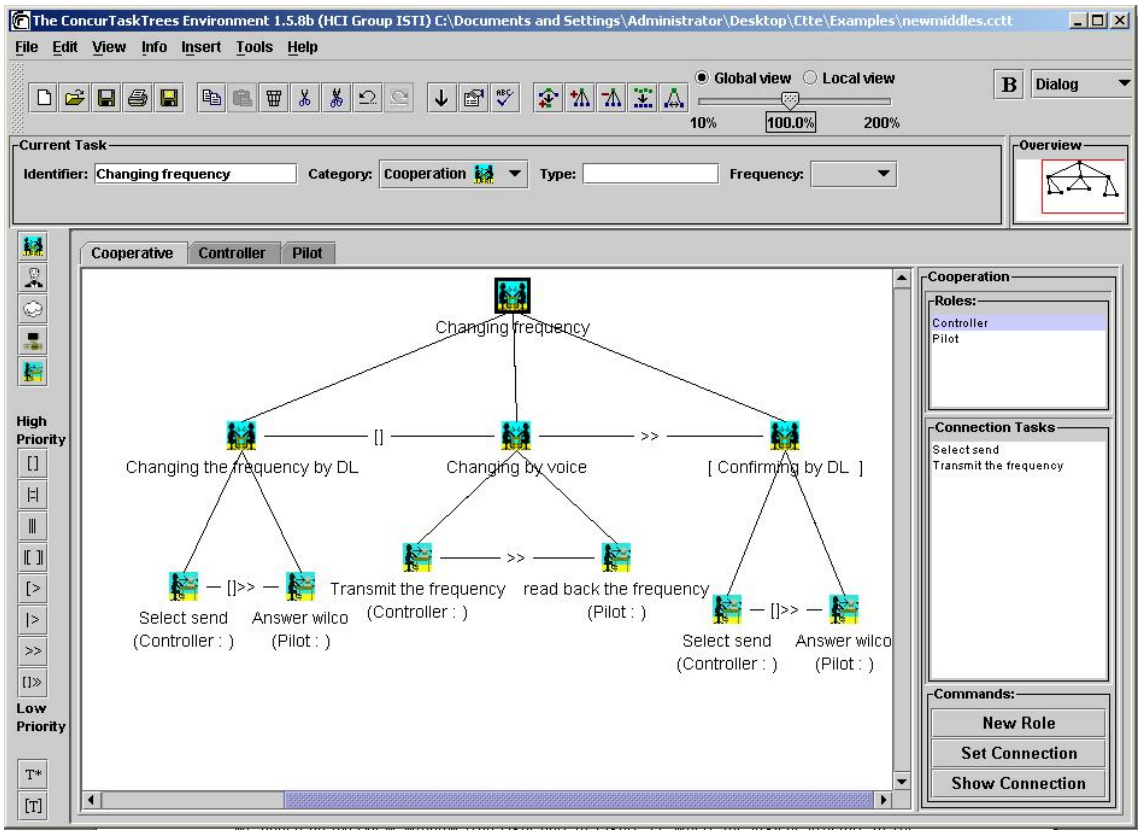

Figure 8. The Editor of CTT Task Models for Cooperative Applications

3.3.2. Simulating Task Models

When we analyse an existing application or designing a new one, it can be rather difficult for the designer to understand the dynamic behaviour resulting from the temporal relationships specified in the task model. The reason is that, especially for real applications, the number of ways in which the application can evolve is high and it is difficult to mentally remember the various temporal constraints among tasks and their possible effects. It becomes important to support a what-if analysis aimed at identifying what tasks are logically enabled if one specific task is performed. To support this analysis of the dynamic behaviour of task models, interactive simulators can be helpful.

A simulator for task models can be useful to better explore the dynamic aspects of task models, including those for cooperative applications. This is a support that only a few tools provide (Biere *et al.*, 1999). Also, in the case of tools for UML, this is a feature usually missing. In addition, CTTE gives the possibility of simulating task models of applications where the resulting behaviour depends on the interactions of various users.

The basic idea is that, at any time, they show the list of enabled tasks according to the constraints specified in the task model. In the list of enabled tasks, only basic tasks, those that are not further decomposed in the model, are considered. Before starting the simulation, the tool

automatically checks that the task model is complete and consistent. The designer can interactively select one of them and the simulator shows, after the performance of the selected task, what the enabled tasks are. This interaction can be iteratively performed a number of times. Since we can describe task models for cooperative applications, in the simulation, we can also see the role of the user involved by the performance of the selected task.

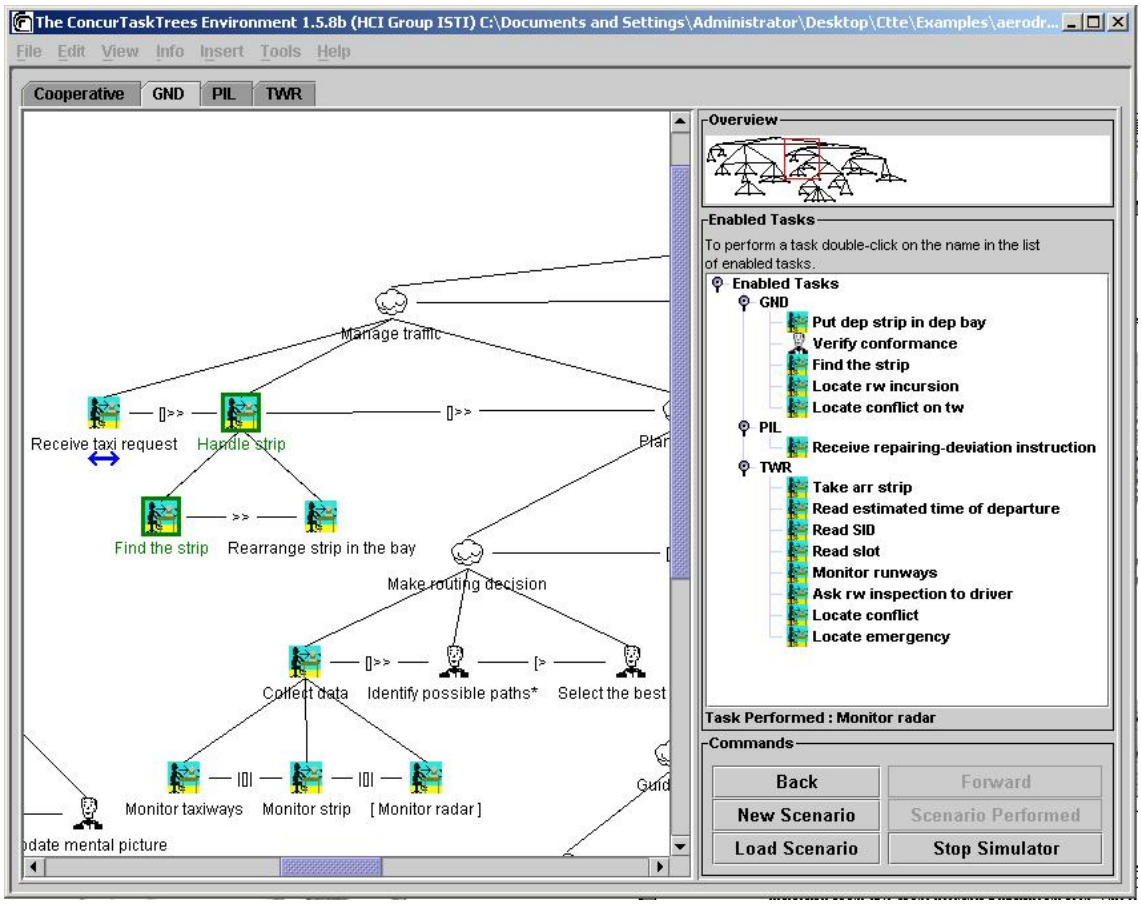

Figure 9. The Interactive Simulator of Task Models

When the simulator is started, then the window on the right displays the list of tasks enabled (see Figure 9). The tasks that appear in such a list are basic tasks, tasks that are not further decomposed in the model. They are grouped according to the role to which they are assigned. In addition, the tasks that are part of cooperative tasks are listed again under the 'Cooperations' label. The enabled tasks are also highlighted in the task model by a green frame.

Then, the user can interactively select a task to perform and the simulator shows what the next enabled tasks are. At any time, it is possible to go back through the performance of the tasks, which means that the effects of the performance of the last task are undone and the list of enabled tasks becomes the same as that previous to the performance of the last task. At this point, the user can choose to go further backward in the task sequence or forward, either through the same path or a different one.

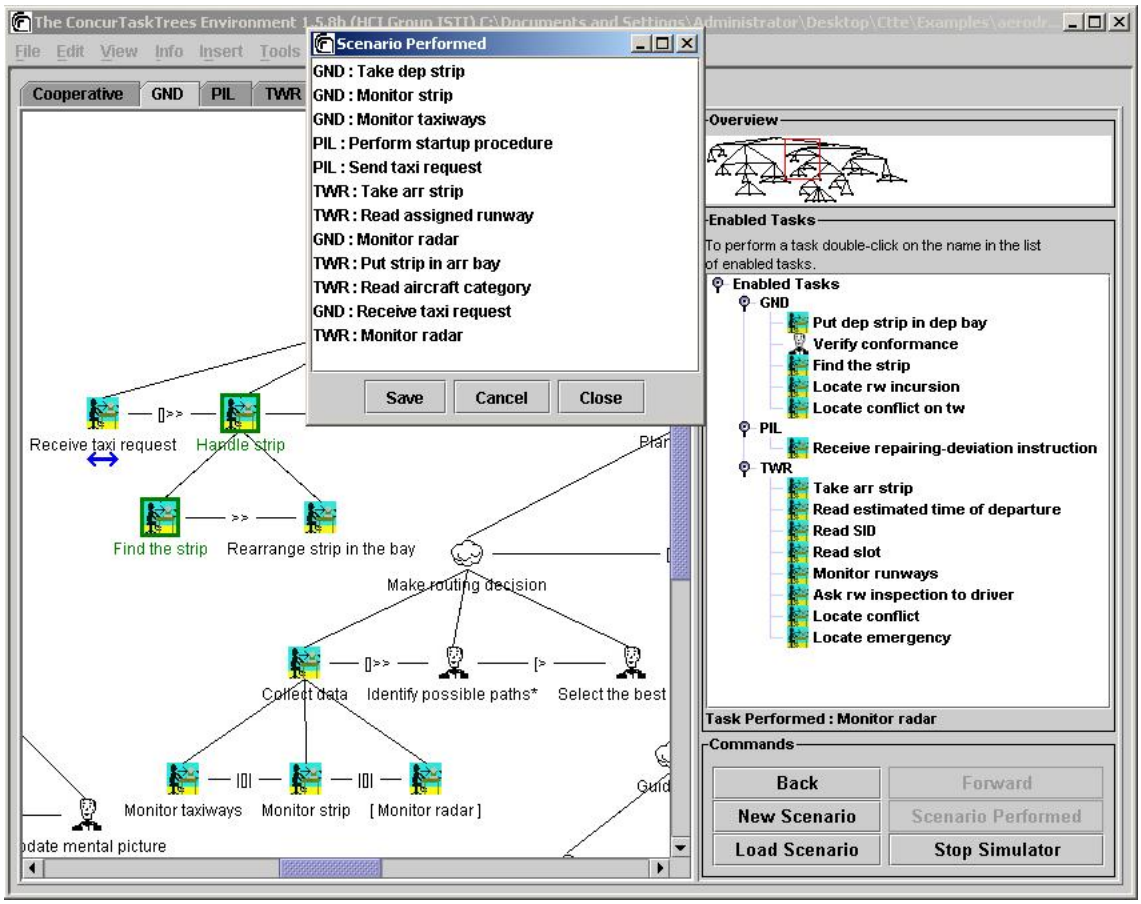

Figure 10. Example of Scenario Derived from Task Model

At any time, the designer can also display the specific sequence of tasks that has been performed in the current simulation (see, for example, Figure 10). They appear with an indication of the role of the user that performs the task. This is a way to interactively identify an abstract scenario that can be saved in a file and used to compare different task models. The tool is able to load a scenario created with another task model in order to simulate performance of the same sequence of tasks. If this is not possible, either because a task is not supported in the other model or because the temporal relationships in that task model do not allow such a sequence, then it means that the scenario is not supported. This can be useful to compare the task model of an existing application with that of an envisioned one or two different task models that are related to two different designs of the same application. Further possibilities are supported when loading a previously created scenario, such as extending it by adding further tasks or exploring variants of that scenario by performing it partially and then choosing different possible evolutions. The simulator has shown to be useful in several cases:

- Designers can check whether the specified behaviour is really what they intended to describe. This is important because, especially in the case of large specifications, it is difficult to immediately understand the overall behaviour deriving from the combination of the hierarchical structure and the temporal operators.
- It can support a multidisciplinary discussion where people with different backgrounds (designers, software developers, end users) discuss design decision at the task level. It can be employed as interactive documentation of an application to explain to end users how to use it (indicating in which order

tasks can be performed, possible choices, and other dynamic information).

In this thesis we will provide some other examples of possible uses of the scenarios produced by the tool. More specifically, in Chapter 6 scenarios will be used as a manner of communicating results of model checking process integrated with the tool., whereas in Chapter 7 they will be used as a means of communication between task models and system models.

3.3.3. Handling Informal Descriptions

Often, it is difficult to immediately create a model from scratch. Thus, to support the initial modelling work, we give the possibility of loading an informal textual description of a scenario or a use case and interactively selecting the information of interest for the modelling work. In this way, the designer can first identify tasks, then create a logical hierarchical structure, and, finally, complete the task model.

To develop a task model from an informal textual description, designers first have to identify the different roles (they are in circle 2 and they are derived from the part of the scenario highlighted by circle 1 in Figure 11). Then, they can start to analyze the description of the scenario, trying to identify the main tasks that occur in the scenario's description and refer each task to a particular role. For example, as you can see from Figure 11, the looking at the taxiways task has been identified within the scenario's description of an example in the air traffic control domain (see circle 3). As this task refers to the ground controller, the Ground role has been selected in the list of roles and the task has been added to it (see circle 4). It is possible to specify the category of the task in terms of performance allocation. In this case, the task has been considered as a user task. In addition, not only a description of the task can be specified, but also the logical objects used and handled (see circle 5). Reviewing the scenario description, the designer identifies the different tasks and then adds them to the list. This must be performed for each user in the environment.

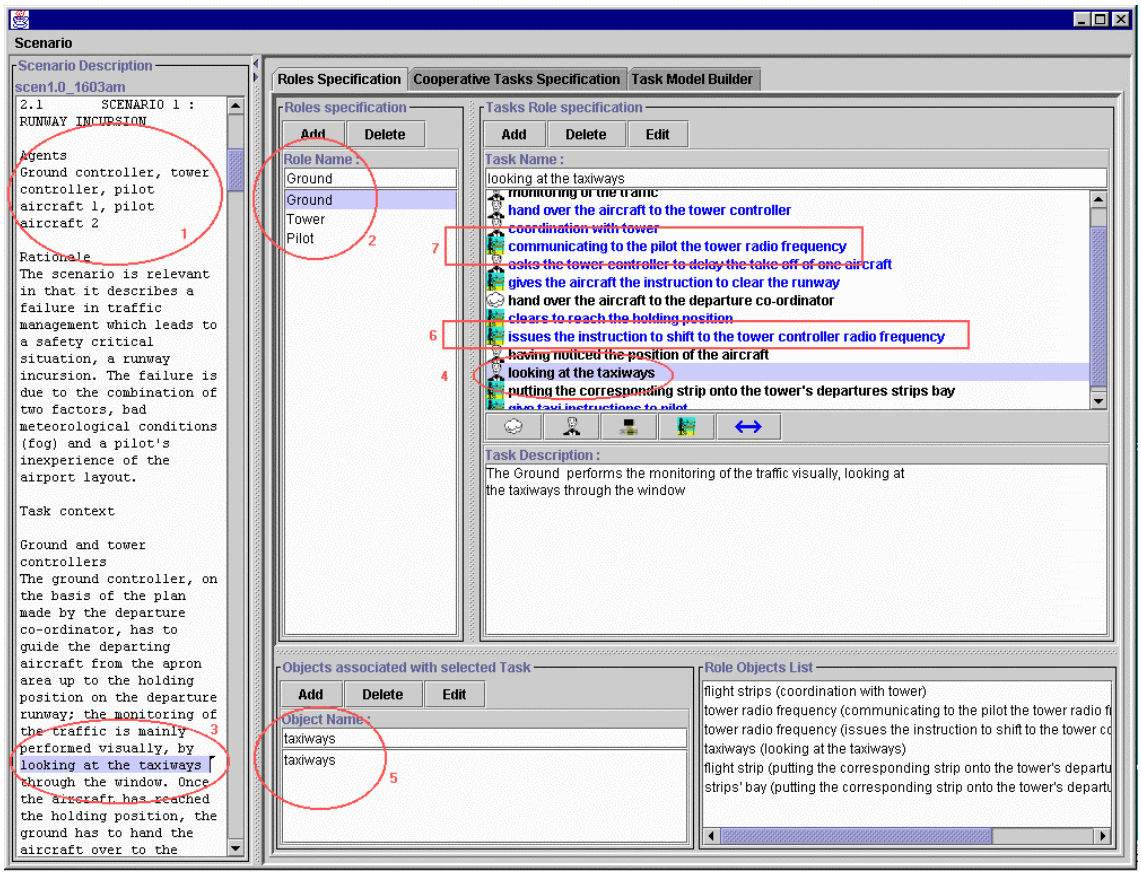

Figure 11. Moving from Informal to Formal

When each user's main tasks in the scenario have been identified, it might be necessary to make some slight modifications to the newly defined task list. This allows designers to avoid task repetition, refine the task names so as to make them more meaningful, and so on. For example, in Figure 11, two similar tasks have been identified: the communicating to the pilot the tower radio frequency and issues the instructions to shift to the tower controller radio frequency tasks (see circle 6-7). They both concern the task of communicating the tower frequency to a departing aircraft that has reached the starting point of the runway, so they can be merged into a single task.

3.3.4. Multiplatform Support

The increasing availability of new types of interaction platforms has raised the need for new methods and tools to support development of nomadic applications, namely the ones that can be accessed through a variety of devices. This has urged the need of specifying whether a task might (or might not) be supported by a specific platform. In fact, for example, phones are more likely to be used for quick access to limited information, whereas desktop systems better support browsing through large amounts of information. The consequence is that a task like seeing a long review on a book or a movie could not make sense on devices like palmtops or mobile phones.

In order to consider such issues we decide to add within the task template a new property, –*platform*– so as to make possible within CTTE, to specify for every task the list of platforms that support its performance. Then, a nomadic application might have tasks supported by one, two or more platforms (see Figure 12), which means that in the 'integrated' task model we could have the specification of all the activities performed on all the platforms considered. However, we usually consider one platform at a time. To this goal, within the tool it is possible to automatically filter the task model according to a specific platform. The tool allows the user to use this feature by means of selecting a specific platform within a window (see Figure 12).

The resulting task model will be a task tree in which only the tasks supported by the specified platform will be represented. Of course, after the automatic filtering process the resulting task model could have disconnected parts. If it is the case, two options are presented to the user: choosing between manually adjusting the task model or letting the tool automatically derive the missing connections according to the priorities of the operators involved.

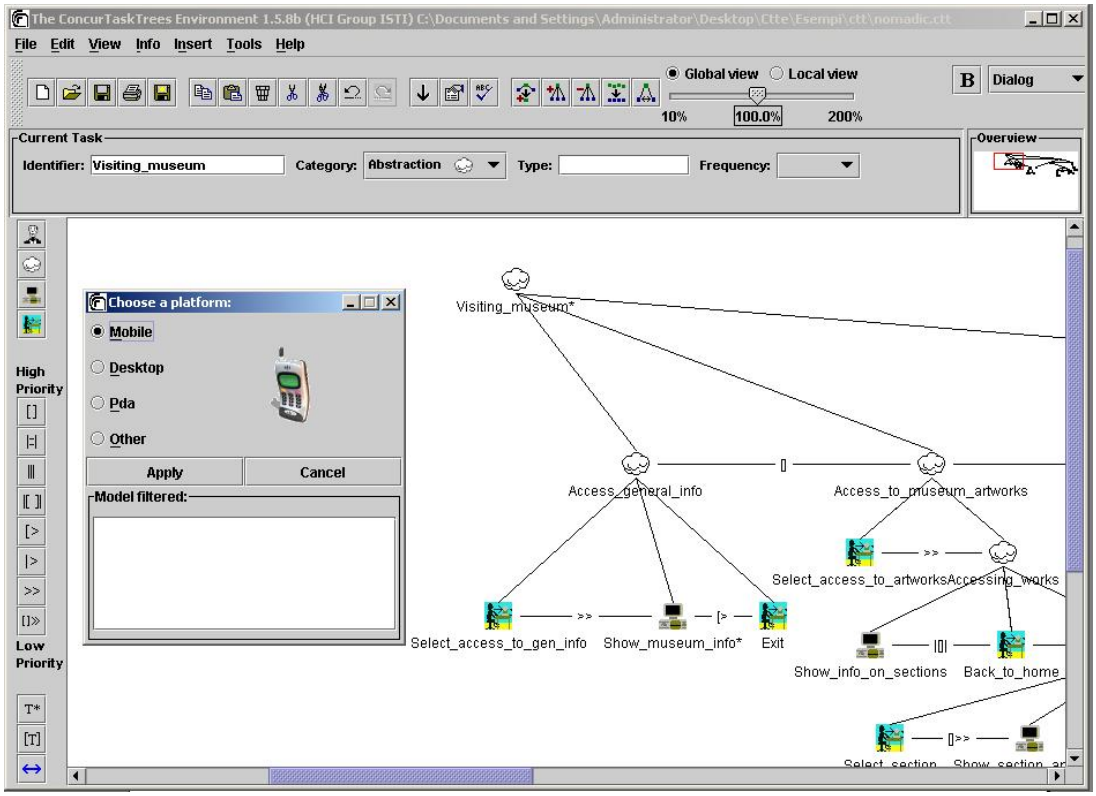

Figure 12: The Window for Filtering According to Different Platforms

3.3.5. Multiple Interactive Views

One usability problem often occurring with tools graphically representing models of realistic systems is that such models do not fit in a window. The usual solution is to include scroll-bars to navigate in the representation. These are useful but often not sufficient to support designers in analysing such models. Thus, to better support the analysis of large specifications, we added an overview window (top right part in Figure 12), where the logical structure of the task model is represented without reporting details, such as the task names or the icons indicating task allocation performance. In the overview window, the part currently represented in the large window is highlighted by a red rectangle. Thus, designers can have, at the same time, a detailed view on a part of interest in the task model that can be edited by direct manipulation and then have an overview on where such a part is located in the overall structure of the model. It is possible to change the part displayed in the detailed view by just selecting another part in the overview window. In addition, it is possible to zoom in/out the view of the detailed representation and the centre of the zooming can be either the centre of the specification or a task selected by the designer.

3.3.6. Checking the Specification

There is also automatic support to check that the specification is complete according to the syntax and the semantics of the notation. This means, for example, that the temporal relationships for all tasks are defined as follows: If a task is a leaf in the cooperative part, then it should appear in a single user part and, vice versa, if a task is defined as a connection task in a single user part, then it should appear in the cooperative part. Another control is that each non-

basic task has at least two children, otherwise it would not be meaningful to have decomposition. Figure 13 shows an example of the results provided by the automatic check. As can be seen, there are errors that stop the user from going any further with the analysis, whereas warnings, which are errors as well (such as a basic task assigned to an abstract category), allow the user to continue with the analysis, for example, by activating the simulator.

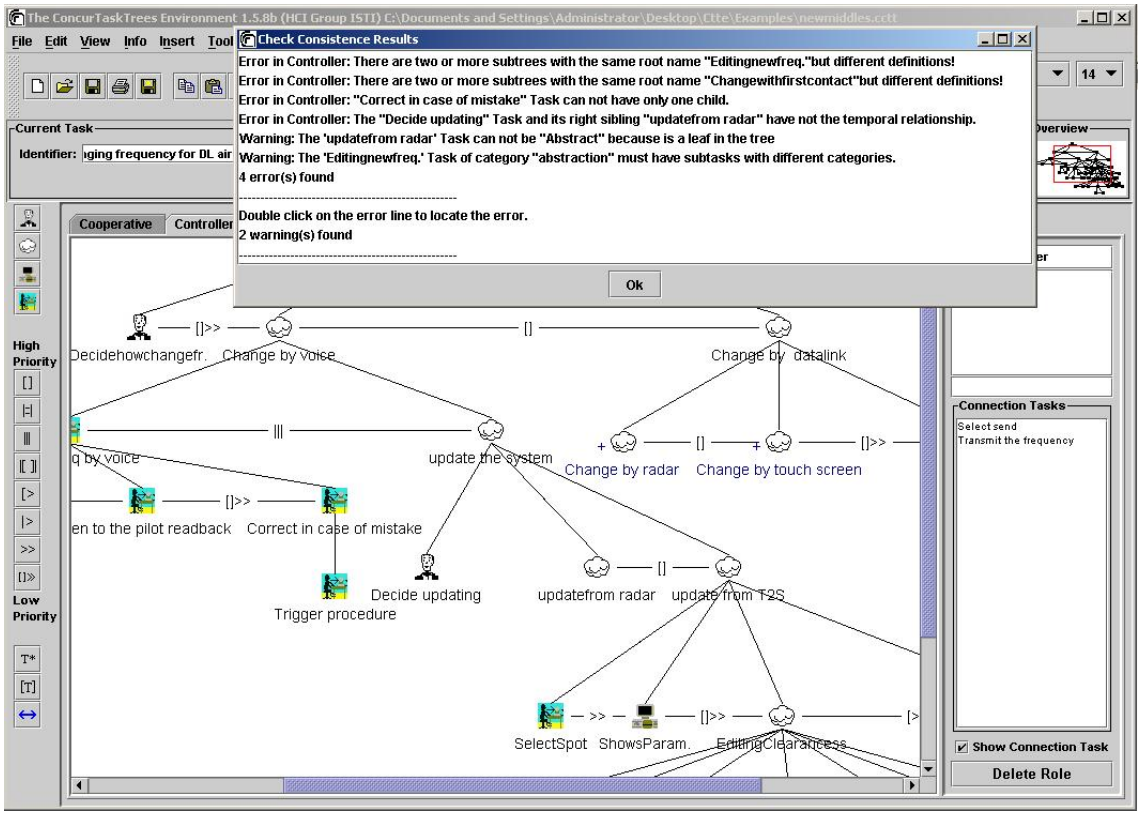

Figure 13. Example of Errors Identified while Checking the Specification Completeness

3.3.7. Comparing Task Models

There is often a need to compare task models. This occurs when people want to compare how people work in the current system and how they could work in a new envisioned system or the designer could be interested in comparing the implications of two alternative designs at task level. CTTE gives some information that can be helpful for this purpose, which is not true in other environments. To be comparable, the two task models should consider the same roles. The comparison is performed in terms of number of tasks, number of basic tasks (the tasks that are no longer decomposed), allocation of tasks, number of instances of temporal operators, and the structure of the task models (number of levels, maximum number of sibling tasks). This information can also be given for single task models in order to analyze them. By comparing this type of information, it is possible to deduce some general features of a solution with respect to another one. For example, a higher number of application tasks and a lower number of user tasks imply that there is a strong shift toward allocating task performance to the system or a higher number of sequential operators imply that the solution supports a higher number of modes in its dialogues with the user.

In Figure 14, there is an example of comparison; designers have to select which part of the task model they want to compare (the tower controllers, TWR, in the example) and then the result of the comparison of the information relative to that part in the two task models appears. There is also the possibility of activating the presentation of the details related to some parameters. For example, if the details of interaction tasks are selected, then the tool shows the

interaction tasks of the selected role that are in one task model but not in the other and vice versa (see, for example, Figure 14, lower part).

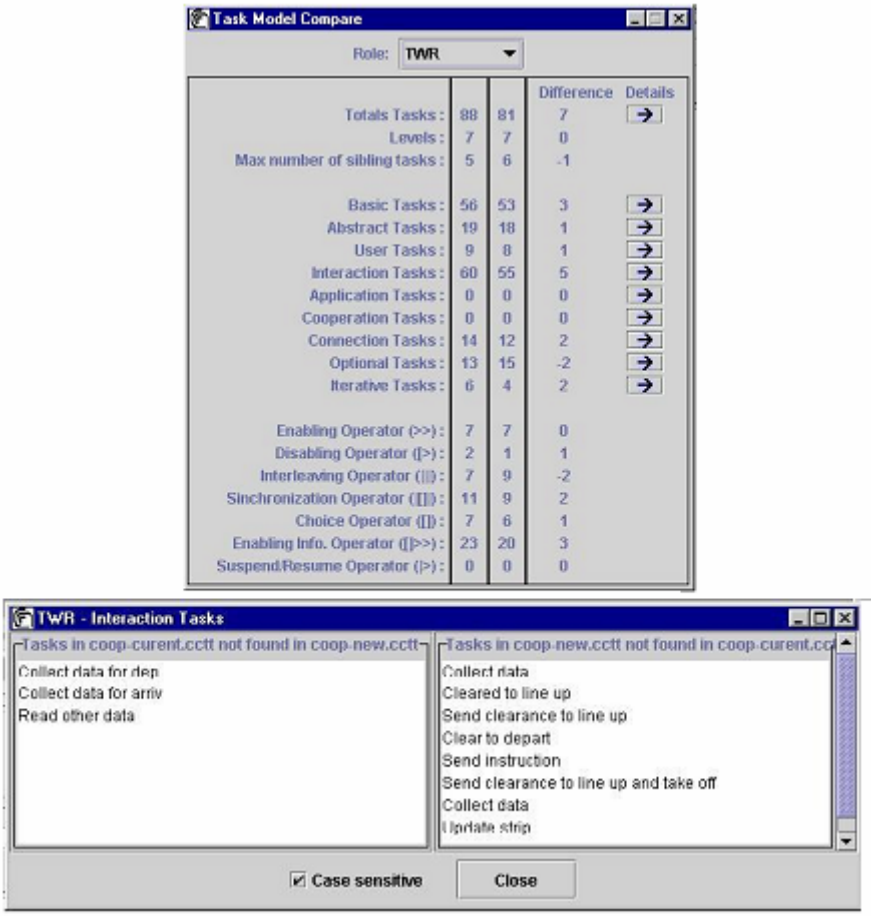

Figure 14. Example of Comparison of Task Models

We would rather avoid having the tool provide definitive interpretations of these results because they often depend on the type of application considered and features that would characterize a good solution in one application domain may represent a bad solution in another. On the other hand, the automatic analysis highlights specific features of the solutions considered that otherwise would have been difficult to identify, especially when large models of real applications are considered.

3.3.8. Additional Features

The tool also supports additional features. We just cite the automatic translation of ConcurTaskTrees specifications into LOTOS specifications (this will be more extensively treated in Chapter 6) for verification purposes. Another possibility of the tool is to automatically calculate Presentation Task Sets, which are composed of tasks that are enabled over the same period of time according to the constraints indicated in the model. This information is useful during user interface design because the interaction techniques supporting the tasks belonging to the same presentation task set should often be part of the same presentation (it will be described more in depth in Chapter 5). In addition, the tool allows to save the information contained in task model in different formats, also in XML format. This improves the interoperability of the tool with other applications.

4. A Task Model-Based Approach for the Design, Verification and Evaluation of Interactive Systems

Summary

This chapter is structured as follows. First an overview of the method that was developed in the MEFISTO Project is proposed. This project dealt with developing novel techniques for the design and the development of interactive systems, when safety and usability are specific concerns This overview introduces the phases and their articulation. After having described such phases, in the last section we will provide the reader with our task model –based approach for the design and development of interactive safety critical systems.

4.1. INTRODUCTION

The use of task models has entered into current practice when developing various types of software product. However, although the use of models has subsequently been applied in the design of interactive applications, we need systematic methods able to precisely indicate how to use the information contained in the model, and this issue is particularly relevant in safety-critical contexts, where human life can be threatened by poor design.

In this chapter, we propose a method for using information contained in formally represented models in order to support and drive the design of interactive applications accordingly, with particular attention to those applications where both usability and safety are the main concern. More specifically, we start presenting a method for the design and the development of interactive systems, when safety and usability are specific concerns. This method has been derived within the Mefisto Project (in which the design of user interfaces for ATC was considered from both a safety and usability point of view, http://giove.isti.cnr.it/mefisto.html) and takes its root in several development methods both from software engineering and human-computer interaction field. More precisely concepts come from, on the one hand, the Merise method (Tardieu *et al.*, 1983), OOAD (Booch, 1994), OOSE (Jacobson, 1992) and more recently UML (Rational, 1997) and, on the other hand, human-centred development (Preece *et al.*, 1994; Dix *et al.*, 1998), prototyping (Rettig, 1994), interactive systems design (Newman and Lamming, 1995).

As you will notice, in such a method task models intervene in a number of phases. Section **Errore. L'origine riferimento non è stata trovata.** will offer a sort of task model -based perspective of such a method. The goal is not only to show more in depth in which way task models are brought into play within the various phases of the lifecycle of the development of an interactive system, but also how the use of the CTT notation and the related tool made possible to exploit specific techniques in the various phases.

4.2. THE MEFISTO APPROACH

This section presents a global overview of the MEFISTO method. The various phases/components of the development process are represented on the left-hand side of Figure 15, while the right-hand side of this picture describes the aim of each phase. The phases are represented in a sequential way from top to bottom, the early phases in the development process being at the top and the later ones at the bottom.

The vertical axis represents the fact that the early phases deal mainly with available information extracted from current tools or current practice while the later ones contains information about envisioned work or artefacts.

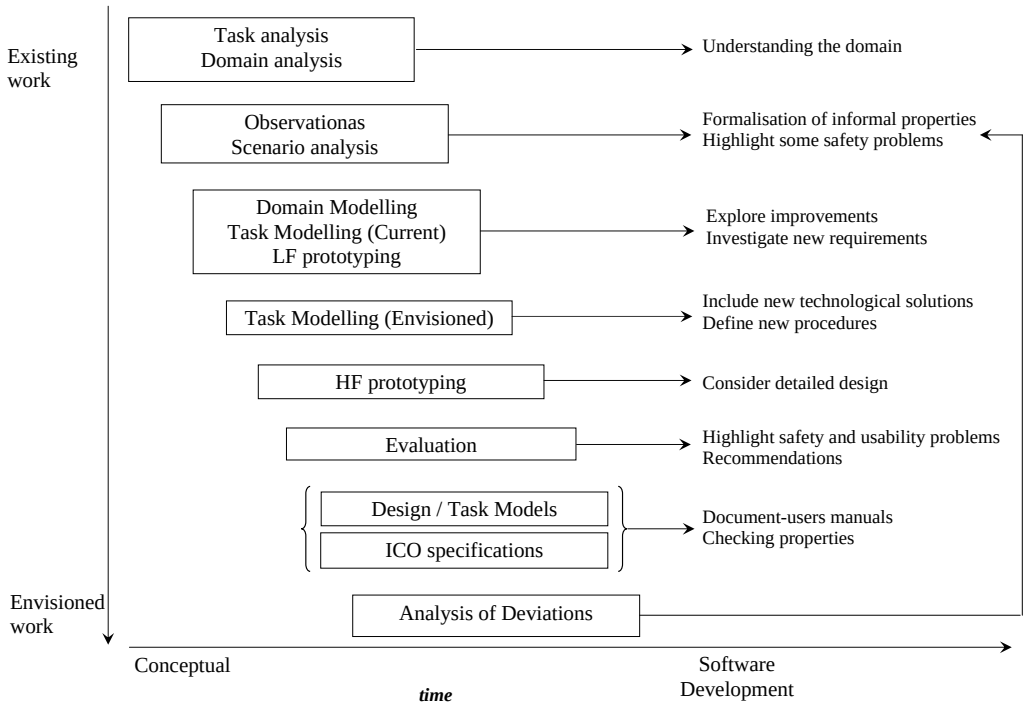

Figure 15: A Sequential View of the Mefisto Approach

The horizontal axis represents at the same time two different dimensions:

- The *temporal aspects*, namely the processing of the various phases according to a temporal sequencing. It is worth pointing out that this is a very global view of the phase evolution, as most of these phases are performed in an iterative way i.e. the output of phases feed other phases even though in the diagram the latter are above the former ones.
- The *abstraction process* describing the fact that early phases deal mainly with abstract information while the later ones deal with more detailed and concrete information. Here again this is only an overview and this abstraction cycle is detailed in Section 4.2.3.

The design process is articulated around three complementary cycles: the *process cycle*, the *design cycle* and the *abstraction cycle*. Each cycle in described in detail, in the following sections, in order to provide the different viewpoints to the development process of safety critical interactive systems.

4.2.1. The Process Cycle

The process cycle describes the path that has to be followed to build both usable and reliable interactive systems. Figure 16 shows a schematic view of the process cycle, articulated into four main phases: observation, design, prototyping and development. As you can note, it does not contain information about the various models that have to be built, because they will be detailed in the abstraction cycle in Section 4.2.3.

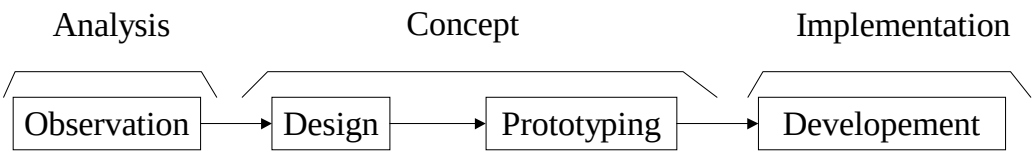

Figure 16: Schematic View of the Process Cycle

The *observation* phase consists in gathering information about the domain. This phase should include observing work practice, existing artefacts, business or organisational environment constraints, etc. The observation phase is crucial for collecting all the relevant information both about the possible problems of use with the current system and the features that work fine. Figure 17 presents a schematic view of the observation phase. The horizontal axis represents the entire domain and activities that have to be studied. As although for real size applications it is usually impossible carry out an exhaustive and detailed analysis of the whole system domain and user activities, the vertical axis represents the fact that the entire domain is only studied in an abstract way, for instance through abstract tasks modelling and abstract domain modelling. For the parts that are defined as critical for the application the study will go more in detail for instance through the selection of case studies that will be fully analysed, and for which all gathered information will be stored in models.

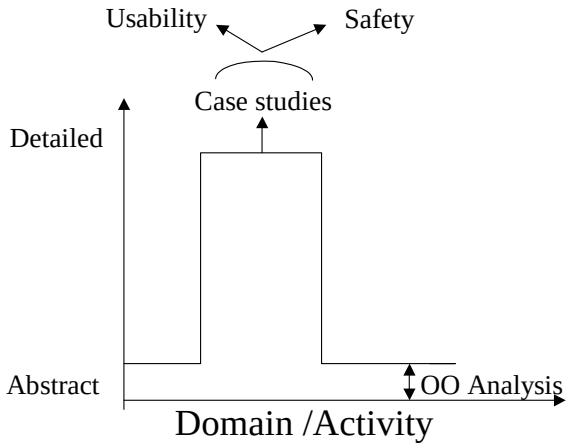

Figure 17. The Observation Phase: Focussing on Relevant Parts of the Domain and Activity

The *design* phase structures and abstracts the information gathered in the observation phase. This phase requires the construction of models representing the information. Analysing and reasoning about the models allows for discovering dysfunctions in the organisation, work practice, tuition of users, artefacts or systems currently used. Lastly the formal models developed in the design phase can be used to generate test cases to be applied to the final system. This will be another way of ensuring conformance between the system built in the development phase and the prototype that has to be developed in a user-centred way.

The *prototyping* phase is split into two separate sub-phases: the low-fidelity prototyping phase and the high-fidelity one.

- The *low fidelity* phase aims at producing precise requirements for the new system to be built. It exploits techniques such as paper and pencil prototyping and requires frequent validation by real users. This phase is supposed to be very iterative and to propose several choices to users. For instance, for each problem to be solved, several design solutions will be built and validated with users.

- The *high fidelity* prototyping aims at producing interactive systems as close as possible to the final system. This prototype is constructed from the low fidelity prototype and will also evolve according to modifications occurring after the validation phases through user testing. With respect to the low-fidelity prototyping phase modification occurs at a lower level of details and thus does not require complete rethinking of the prototype. However, due to these iterations the software quality of the hi-fidelity is rather low and it thus cannot be used as the final system. This is even truer in the field of safety critical interactive systems. Another important aspect is that the final high-fidelity prototype built in the prototyping phase is to be used throughout the development process as a source of information and design solution.

The *development* phase aims at producing high-quality software for an interactive safety critical application, which should meet all the requirements that have been identified in the previous stages of the process.

4.2.2. The Design Cycle

The aim of this cycle is to show how the various main activities fit together: as the development process of interactive systems is by nature iterative, it is not by chance that the activities have been represented in a circle. Within these circle (see Figure 18) four quadrants represents the four main activities: task modelling, system modelling, prototyping and user testing.

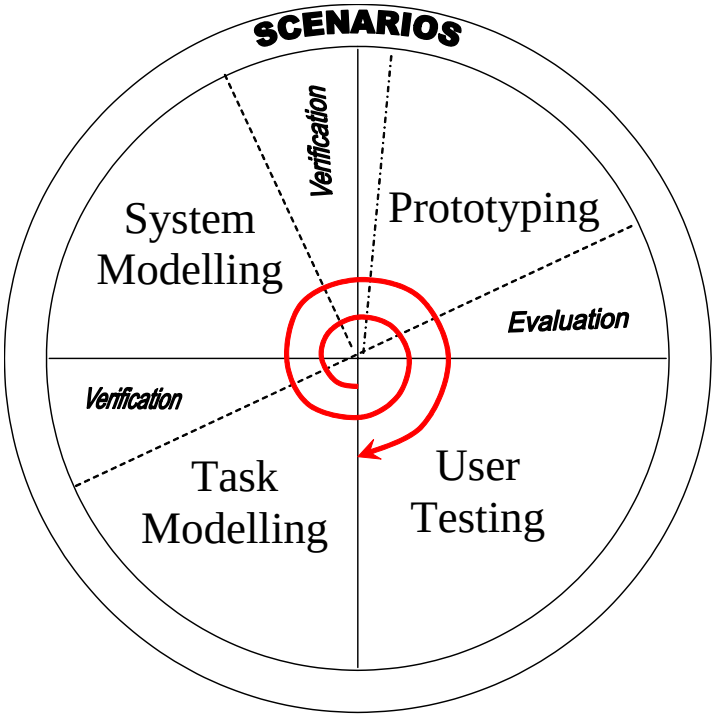

Figure 18. Detailed Iterative Design Process

The spiral lines at the centre of the diagram represent the fact that the process should start with a task analysis and modelling phase, while the external circle labelled “Scenario” describes the fact that scenarios are used in all the phases of the cycle. They can be used to drive user testing or as the starting point of the task modelling activity. They can also be used as a way of checking the system model.

4.2.3. The Abstraction Cycle

This cycle describes the various models that should be built following the method while designing safety critical interactive systems. For each of the boxes in Table 3 MEFISTO project selected dedicated notations and tools. CTTE allows for description of informal scenarios, and to derive task models from informal scenarios, describe task models, simulate and verify them. Petshop allows for edition of High-level Petri nets and editing, execution and verification of ICO models.

Levels and Information in the Abstraction Cycle

Basically, three levels of abstractions are addressed and represented through appropriate conceptual models while applying the MEFISTO method. Each level of abstraction allows to have a specific interpretation of the information considered, and in Table 3 they are basically represented in different rows, where the most abstract are at the top and the most concrete are at the bottom.

The upper level is the most abstract and represents conceptual information that is unlikely to change frequently and is quite invariant in the domain under consideration. Information at this level is independent from the organisation of work or the artefact used. While building those models the designer must only provide information corresponding to the question "what". Information corresponding to question such as "how**"** or "who**"** must not be represented in the models of the conceptual level. The middle level represents information independent from the implementation but more concrete than the ones in the conceptual level. They typically correspond to answers like "who does what when and where". Information such as how things are performed must not be captured in the middle level. This level highly depends on the organisation of work or on the artefacts that are put at users' disposal. The lower level represents concrete and precise information. Such information corresponds to the answer "how". They are thus likely to change quite frequently with artefacts or work practice.

	Objects	Context	Activity	
CONCEPTUAL LEVEL Concepts	Domain level	High Level Scenarios Stages	Goals	Functional goal What the organisation does Business and Safety constraints and regulations
ORGANISATIONAL LEVEL Information	Domain model	Roles and Agents Environments	Task Models	Architecture What does What, When, and Where
PHYSICAL LEVEL Details	Object-Oriented Programs	Stages Scenarios Instances	Practice	Physical How to perform actions

Table 3: Content and Structure of the Abstraction Cycle

In addition, in each column in the matrix gives a particular view on the application. The first column is more concerned by the system/software viewpoint and thus gathers information about the objects (both at the domain level and the application level) or processing. Information in this column are required for building the application and mainly concern software engineers (even though they can also be used in the other columns). The second column deals with contextual information that needs to be collected in order to understand the context in which artefact will be deployed and the context in which users' activities will take place. The last

column deals with users' activity. At a high level of abstraction it represents users' goals. At the organisational level it represents the users' tasks and the activities they carry out in the organisation. The lower level is more concerned with real and concrete practice.

4.2.4. The Sun Curve

In this section, we will show that the abstraction follows a Gaussian curve (being concrete at the beginning and at the end of the design process and being more abstract in the central phases), and can be represented through a sun curve (Figure 19). The sun curve derives its name from the fact that it resembles the perceived trajectory of the sun in the sky. Its purpose is to describe the relationship between the phases of the process cycle and the level of abstraction of the information gathered or produced. It starts first with analyses of concrete information from existent systems and activity, then follows a process of abstraction in which modelling is a critical activity and lastly comes back to concrete information with the design of the improved new system.

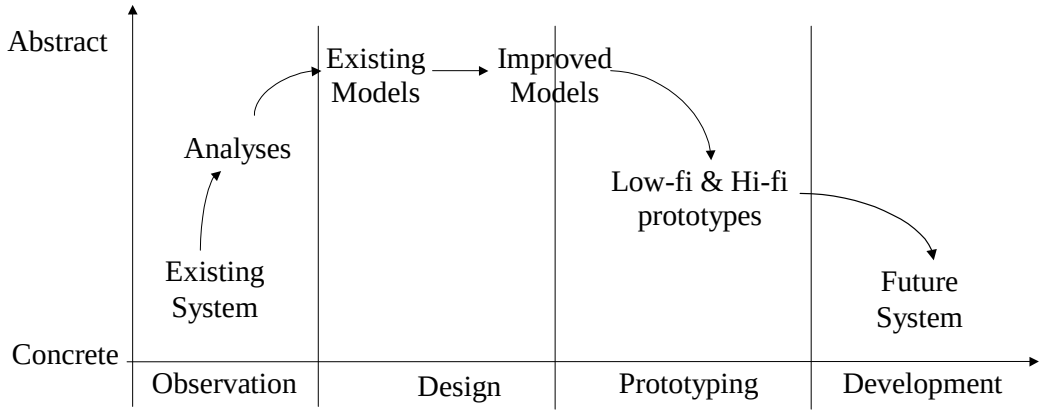

Figure 19. The Sun Curve Describing Level of Abstraction for the Phases

As described in Figure 19, the prototyping phase is a critical phase that relates the design abstract work to the development of the system. It is thus the key element for an efficient transition between abstraction and implementation, and also a phase where user involvement will take place through usability evaluation.

4.3. OUR APPROACH

Along the lines traced by the MEFISTO method, we want to present now our approach about the use of task models within the development process of an interactive safety-critical system. The main goal is both shifting the focus on the role played by task models in such a process, but also highlighting how the use of the CTT notation and the related tool made particular effective to exploit specific techniques adopted in the various phases.

The main ideas of such an approach are highlighted in Figure 20, where the activities have been drawn with ovals and the results with rectangles. As you can see, the 'Task Model' rectangle has been highlighted with a thicker border, and the arrows departing from it as well, also labelled with different letters, in order to refer more easily to the subparts of the framework that receive task models as an input.

The process starts with a requirements elicitation phase in which user requirements are derived and represent the input for the both the task modelling and system modelling phase. The result of task modelling phase is a task model in which the activities that are supposed to be performed have been described. The information contained in the task model can be used in several ways. It can be compared with the information contained in the system model, in order to check if the two models represent the same system or, alternatively, there is any inconsistencies that might trigger some modifications on the task modelling side, or the system modelling side, or both.

Also, we envision that as soon as a task model is available, a first verification phase can take place, with the goal of identifying any incorrectness in the specification early in the process, which is highly recommended and justified by the fact that we are mainly dealing with safety critical applications. Such an early evaluation of user interfaces is carried out by means of formal specifications, with the main benefits that claims can be defined very precisely and determined objectively and automatically. If formal models are used for usability evaluation purposes, the challenge is to determine meaningful properties of the model that are valid indicators of some properties of the system (e.g. in case of safety critical systems, usability and safety).

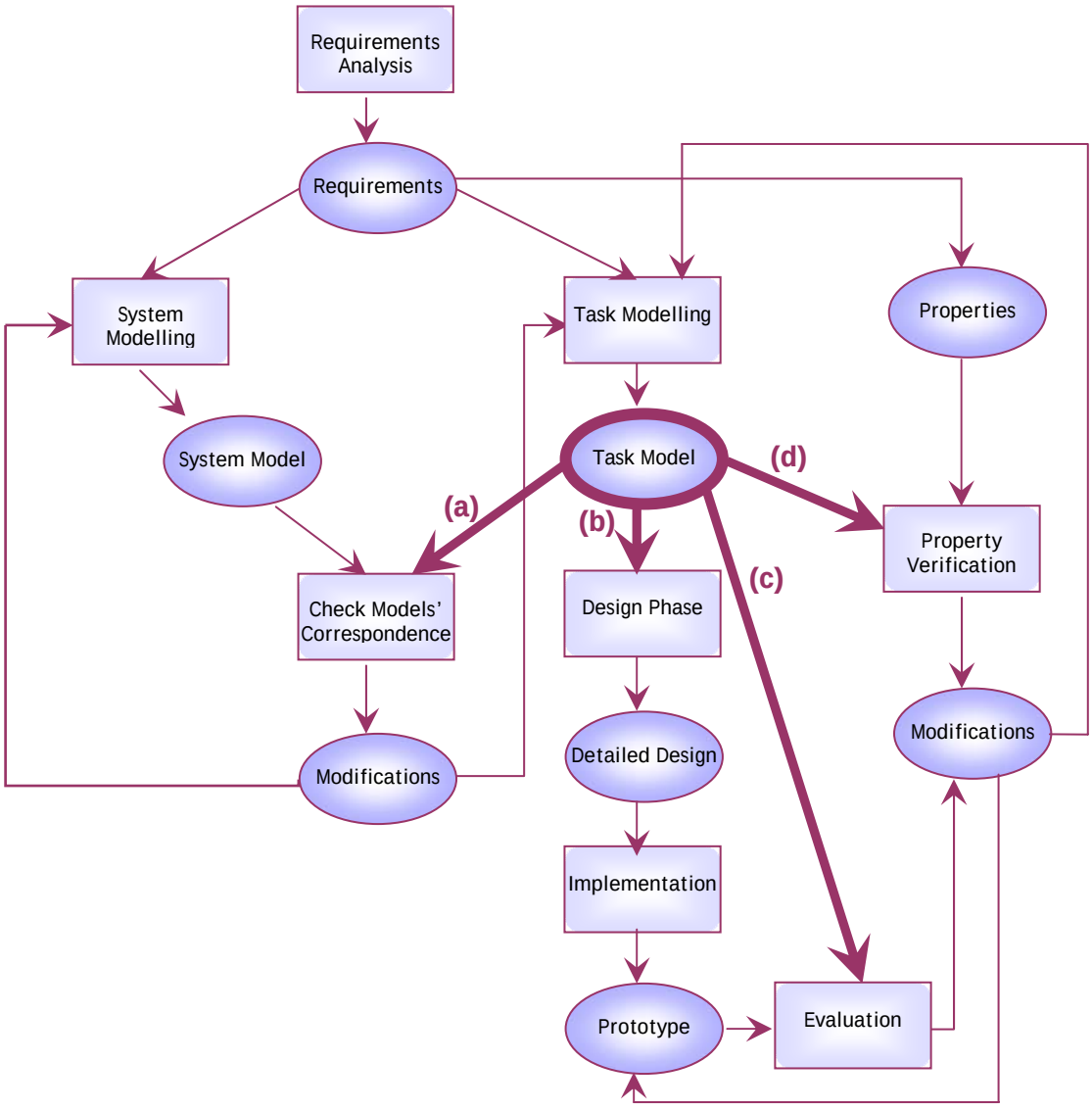

Figure 20: An Overview of the Approach Proposed

The task models might also be used to support a detailed design of the system to develop and, as a matter of fact, it is the input for deriving a detailed design of the user interface, which might in turn trigger an implementation phase in which a prototype is developed. Such prototype might be used, together with the information contained in the related task model, to perform an evaluation step where the effects of possible deviations from the activities specified in the task model are assessed. The benefits of carrying out this kind of evaluation on a specific

prototype might also improve the knowledge developers have of the system itself, both for documentation purposes and, of course, for driving recommendations for an improved future design. Every possible suggestion derived from this process is used to modify the task model specification or the prototype and iterate the process.

The picture in Figure 20 is a very high level one, whose main goal is to provide an overview of the approach. Actually, most of the phases need further refinements and descriptions, and it is just what will be done in the rest of the thesis. More specifically, next chapter will concern on how to use task modes to drive the design of user interfaces, Chapter 6 will focus on early verification and evaluation techniques that can be pursued as soon as a task model is available, Chapter 7 will highlight the importance of integrating information contained in task models with that contained in system models, while Chapter 8 will detail a specific inspection based evaluation technique that might be effectively adopted especially when dealing with safety-critical applications. The last chapter will provide examples derived from a case study in the ATC domain in which our approach has been applied.

5. Task Model -Based Design of Interactive Systems

Summary

In this chapter we describe how the information contained in task models can support the process of designing an abstract description of the related user interface defined in terms of abstract interaction objects mainly identified by the task they support. The dialogues of the user interface are also derived in such a way that temporal constraints will be as closest as possible to those specified in the task model. More in depth, we describe both such an abstract language, and the rules that we identified to progressively obtain such a description from the related task model. The abstract description of the user interface is then made concrete through appropriate transformation that take into account the specific environment at hand. An example of application will illustrate and exemplify the main ideas of our approach.

5.1. INTRODUCTION

One of the most difficult phases in the development process of an interactive system is bridging the gap between the conceptual analysis phase to the more proper design phase of the user interface that should meet user requirements. Despite the availability of several off-the-shelf software packages aimed at reducing such a gap, the success of this process often remains too dependent on the particular expertise of designers, who are put to the test both when they have to identify which interaction object is more suitable to implement a specific interaction, and when they have to combine such interaction objects to develop complex user interface systems from scratch. So, a methodology is required, able to operatively support a user-oriented, semantics-based perspective of the design of user interfaces, helping designers manipulate and reason about more 'abstract' descriptions of the interactive system to develop.

As a matter of fact, declarative representations or 'abstract' vocabularies of interface elements characterised in terms of the intention they support rather than focused on low level details about their physical appearance have been more and more emerging as a useful means to cope with this problem. One example can be found even with XForms (W3C, 2003), a new generation of Web forms whose main idea is separating application data and logic from mere physical presentation-driven aspects, to the aim of making the Web robust with respect to accessibility, internationalisation, evolution of technology, so coping with the test of time. The XForms user interface design creates an XML-based vocabulary that makes user interface controls presentation-independent, lending itself to *task-based authoring* of user interaction that encourages authors to fully specify the intent behind a given control, rather than soon hard-wiring it with a particular representation in a defined environment. In this way, the resulting vocabulary is more relevant, because it expresses the tasks that the various interaction techniques should support, rather than just their rendering attributes.

In this chapter what we propose is just an approach to obtain an *abstract, goal-oriented* description of an user interface (or an *abstract user interface*), starting from the related task model that describes the activities supposed to be performed through the interactive system. Then, we will focus on two main abstraction levels of an interactive system:

- the *task and object model*, where the logical activities that need to be performed in order to reach the users' goals are considered along with the objects that have to be manipulated;
- an *abstract representation of the user interface*, where, instead, the focus shifts more specifically to the interaction objects supporting task performance.

To exemplify such abstraction levels we can consider the activity of booking a flight. At the task level this activity can be decomposed into two subtasks of selecting departure town and selecting arrival town, and optionally, selecting additional seat/meal preferences. When we consider the abstract user interface, we are wondering about identifying the interaction objects needed to support such tasks. Then, for specifying departure and arrival towns we need

interaction objects that, in abstract terms, should allow the user to select an object from a predefined number of possibilities (e.g.: the departure town is chosen from the set of cities where the airline services operate). Such choice, at this level, should give additional information (although in rough terms) about the cardinality of the set from which the selection will be performed, useful information when we move on the concrete level, where we need to consider the specific characteristics of the device at hand. It is worth pointing out that this kind of reification approach is extremely effective when different platforms are considered, although it could be also used in homogeneous environments (from the viewpoint of available platforms).

In this chapter, we propose a method for using information contained in formally represented task models in order to drive the design of interactive applications in a way that should as much as possible coherent with the information contained in task models. The main idea is to use information contained in task models to identify the interaction and presentation techniques best suited to support the tasks at hand and the dialogues whose temporal constraints are closest to those of the model. This is achieved firstly with a task analysis and modelling which is then translated into a hierarchy of objects, where the available interaction objects are classified by their semantics rather than by their appearance. In particular, our analysis will focus on two aspects: how to use the temporal operators among tasks to design and implement the dialogue of the corresponding user interface and how to analyse tasks and their attributes to identify suitable presentation techniques supporting their performance. Some criteria for using information contained in task models for designing the different parts of an UI (the dialog, the presentation and the functionality) will be illustrated in the chapter. A simple example will be also described in the last section in order to give the reader the flavour of the approach, whereas in Chapter 9 a more detailed one will describe more thoroughly how the concepts delivered have been applied in the context of an ATC application.

5.2. RELATED WORK

The design of an interactive system usually includes three main parts: *functionality*, *dialog* and *presentations*. Functionality includes the functional actions and objects that will be available to the user, the dialog concerns the navigational structure and dynamic behaviour of the interface, while the presentation part deals with arrangement and rendering of the objects within the user interface, so for example with graphical user interfaces it will concern about layout and spatial arrangements issues. All three activities are dependent on each other and they need to be kept consistent in order to form a coherent whole. Moreover, some priorities might be identified, for instance if the underneath functionality has not designed well, the system will be not helpful to users and consequently dialog and presentational aspects will soon become irrelevant and, in the same way, the dialog has to be good enough before presentational aspects will be significant.

Also, the problem of designing interactive systems can be described at different abstraction levels, a conceptual one, where the main functionalities and aspects are identified; and a more ‘architectural’ one, where the basic components and their relationships are identified in order to carry out the functionalities. As far as the conceptual models are concerned, the first one that can be cited is the Seeheim model, where there is the just mentioned classical subdivision of user interface systems into three layers: application, dialog, presentation. This approach seemed soon appropriate but too generic, and a number of refinements of this model were proposed, the first one being the Arch model.

Within Arch, the application part of the Seeheim model is further refined into *domain specific* and *domain adapter*. The domain specific part controls, manipulates and retrieves domain data and performs other domain related functions, whereas in domain adapter are implemented domain-related tasks required for human operation but not available in the domain specific component. *Dialog* is the classical part in charge of the task level sequencing. Then we have a *presentation* part, which includes a set of toolkit-independent objects, and lastly the *toolkit* part implementing the physical interaction with the end user.

In the Lisboa model we have the user interface system refined into three kinds of objects: *interaction*, *transformation*, and *monitor objects*. Interaction objects allow the user to interact with a specific media, transformation objects allows for controlling interaction objects and define their behaviour, whereas monitor objects monitor relations and transactions of interaction objects. Finally, the *functional core* is identified, where the functionalities independent from the media and the interaction techniques used to interact with the user are considered.

As far as the architectural models are concerned, we mainly cite three ones: MVC, PAC, and the interactor model. In MVC three parts are identified; model, view and controller. The model basically represents the functional core, whereas the view is the perceivable behaviour for output and the controller is the perceivable behaviour for input. Then, basically, the view and the controller covers the user interface of the objects, namely the behaviour that can be perceived by the user.

In PAC three facets are identified for each interaction object: abstraction, control and presentation. The abstraction part is the functional core, which implements some expertise in a media-independent way so that there is an abstract descriptions of the objects to provide to the users. The presentation part models the perceivable input and output behaviour, the appearance and the reception of the user input The control part links an abstraction to a presentation, controls the behaviour of both abstraction and control, so basically maintains relationships and dependencies between them. This means that in this model, for instance, the presentation cannot communicate directly with abstraction and vice-versa.

Another attempt to express the architecture of an interactive system was presented in (Paternò, 1994), by introducing the *interactor model*. This model differs from other models (e.g. PAC) because it structures each interaction object as an entity to support bi-directional communication between users and software applications, in particular input from the user towards the functional core and output from the functional core to the user. In this model a number of attributes are listed in order to precisely identify the characteristics of an interactor. They include for instance the *list of tasks* and *objects* associated with the interactor, *first action* (the initial possible actions of the interactor), *last action* (the last possible actions), etc. In addition, a number of composition operators are identified in order to combine instances of such interactors, basically along the two (input and output) flows of information. The supposed goal is to model the fact that an interactor can receive input from other interactors and not only from the user or the application.

As you will see more in detail in the next sections, among such approaches, we mainly got inspiration from interactor model, although several features of the approach are different. For instance we as well define composition operators but, differently from modelling the input/output communication flows of the interactive system, such operators will basically be used to convey information regarding the layout of the different objects within the abstract presentation. In addition, a lot of properties are not considered anymore (e.g. first action, last action, ..) and the dynamic behaviour of the user interface is managed in our proposal at the level of each abstract presentation.

5.3. A PROPOSAL FOR AN ABSTRACT DESCRIPTION OF USER INTERFACES

In this section we describe our proposal for an abstract user interface specification able to describe the architecture of an user interface at an interactor-based level[1]. According to our approach, an abstract user interface is composed of a number of abstract presentations and several connections among these presentations. Each abstract presentation defines a set of interaction techniques perceivable by the user at a given time, while the connections define the dynamic behaviour of the user interface, which means that they indicate what interactions

[1] In the continuation, we will use the expression ‘interactor’ as a synonym for ‘abstract interaction object’.

trigger a change of presentation and what the next presentation is. Such connections will be associated with conditions in case a specific combination of interactions should trigger the change of presentation.

The structure of the presentation is defined in terms of interactors composed through a number of composition operators (see Figure 21 for a class-diagram describing the structure of the dialog). As you can see from this figure, the specification of the interactors is in the shape of a hierarchy, with objects at the lower levels inheriting properties of the objects at the higher levels. The advantage of deriving interactors by successive inheritance is in that it enables exploring the underlying design space of the most suitable interaction objects to support the current task.

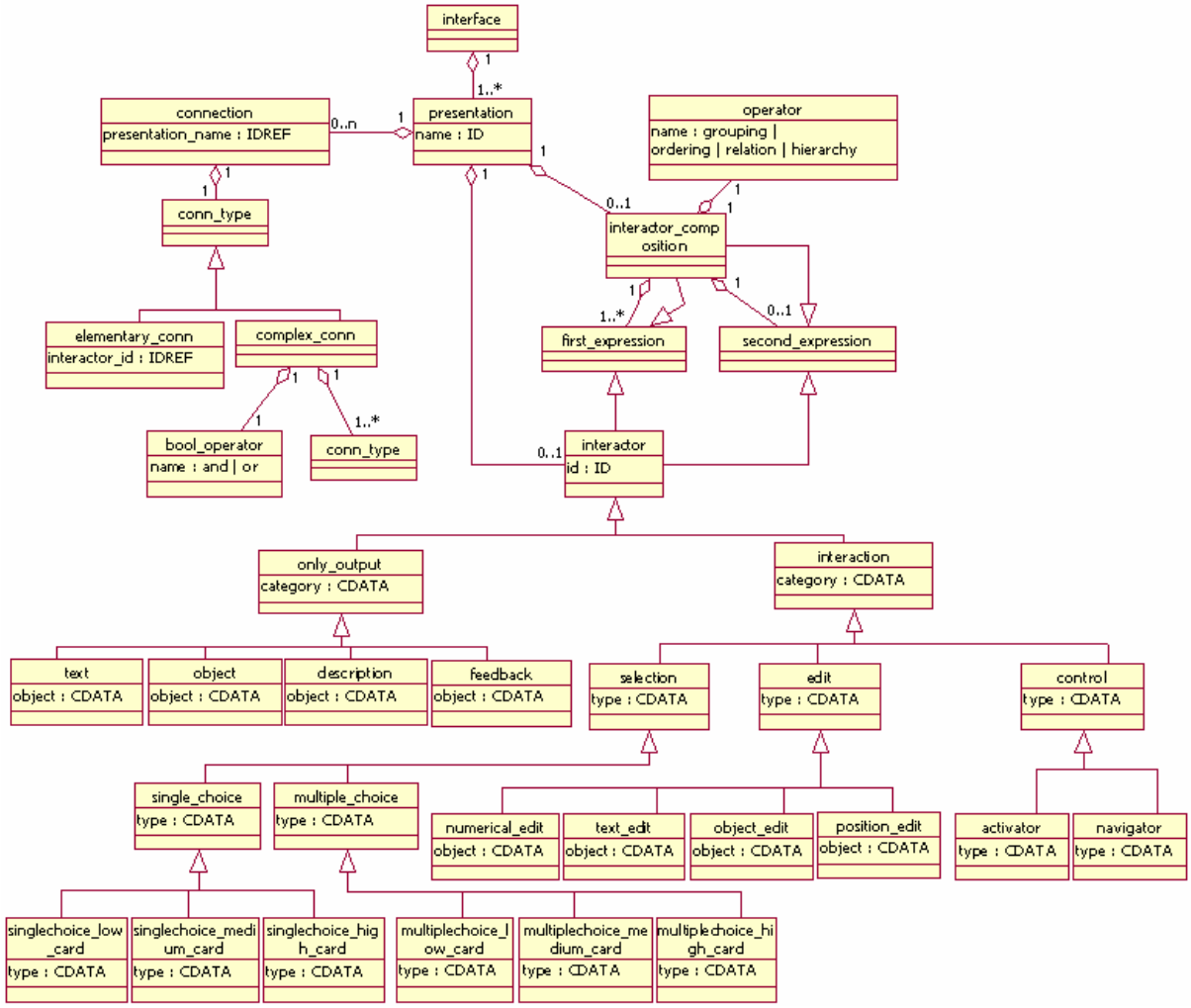

Figure 21. Concepts and Relations for Describing Abstract User Interfaces

Among the different categories of interactors (see 'interactor' class in Figure 21), it is possible to distinguish between those implying an interaction between the user and the application (*interaction* elements) and those implying just a processing action from the application, so they just provide the user with the results of such processing (*only_output* elements). As far as the *interaction* elements are concerned, there are different types of elements depending on the type of task they support: we have *selection* elements (selecting between a set of elements), *edit* (editing an object), *control* (triggering an event within the user interface, which can be useful to activate either an application functionality or the transition to a new presentation). As for the *only_output* elements, they might assume different types (text, object, description, feedback), depending on the type of output the application provides to the user: a textual one, an object, a description, or a feedback about a particular state of the user interface.

The composition operators can involve one or two expressions, and each expression might be composed of one elementary interactor, several ones, or, in turn, compositions of them (see *interaction_composition* class in Figure 21). The operators have been defined taking into account the type of communication effects that designers aim to achieve when they create a presentation (Mullet and Sano, 1995). They are:

- *Grouping* (G): indicates a set of interface elements logically connected to each other;
- *Relation* (R): highlights a one-to-many relation among some elements, one element has some effects on a set of elements;
- *Ordering* (O): some kind of ordering among a set of elements can be highlighted;
- *Hierarchy* (H): different levels of importance can be defined among a set of elements.

In the following sections we will explain the rules that we identified in order to concretely translate the task-based specification into an interactor-based specification.

5.4. OUR APPROACH

In this section we propose our approach for obtaining an *abstract* description of an user interface (or an *abstract user interface*), starting from the related task model describing the activities supposed to be performed through the interactive system. Then, we will focus on two main abstraction levels of an interactive system:

- *tasks and objects level*, where the logical activities that need to be performed in order to reach the users' goals are considered along with the objects that have to be manipulated;
- the level of an *abstract description of the user interface*, where, instead, the focus shifts more specifically to the interaction objects supporting task performance.

To exemplify such abstraction levels we can consider the activity of booking a flight. At the task level this activity can be decomposed into two subtasks of selecting departure town and selecting arrival town, and optionally, selecting additional seat/meal preferences. When we consider the abstract user interface, we are wondering about identifying the interaction objects needed to support such tasks and this identification is carried out in abstract, goal-oriented terms. Then, for specifying departure and arrival towns we need interaction objects that, in abstract terms, should allow the user to select an object from a predefined number of possibilities (e.g.: the departure town is chosen from the set of cities where the airline services operate). Such choice, at this level, should give additional information (although in rough terms) about the cardinality of the set from which the selection will be performed, useful information when we move on the concrete level, where we need to consider the specific platform at hand.

Then, the 'abstract' connotation of the aforementioned description stems from the fact that the focus is now on the specific abstract interaction objects (aio or interactors) used to support the concerned tasks, more than the tasks themselves. The interactors are then the means by which a component can mediate information between the user and the interactive system. Therefore, the main goal will be to identify, through the related task information, the application semantics of each interaction object, leading to a task-oriented classification of interaction objects that facilitates designers and developers to obtain an interactive system supporting user tasks. The different levels that we have just specified call for proper transformations able to move from the different levels still maintaining the semantic relationships across them.

Such transformation will take into account CTT tasks and temporal dependencies existing among them, together with task-related properties, and then it will derive the most suitable interaction techniques capable of appropriately supporting the specified activities. More in depth, the translation between the task level onto an abstract user interface is articulated into a number of steps referred in Figure 22:

- *Identifying PTS and Transitions:* The CTT task model is translated into an equivalent representation expressed in terms of so-called *Presentation Task Sets* (PTS), sets of tasks enabled over the same period of time according to the constraints indicated in the task model. Each PTS then identifies the group of activities that should be supported by each abstract user interface presentation at the same time. The conditions allowing moving across PTSs are also derived from the task model, and called *transitions*. This phase, referred as (a) in Figure 22, is completely automatic and supported by CTTE.
- *Applying Heuristics:* In order to reduce the number of PTS which, in some cases might be very high, a number of heuristics can be applied. The application of the various heuristics (see (b) in Figure 22) is supposed to be triggered by the designer, while the CTTE allows the automatic calculation of the resulting sets.
- *Mapping PTS-Based Specifications into Interactor-Based Descriptions*: In this step, the tasks included into the various PTSs are translated into suitable groups of interactors supporting the performance of such tasks, and such groups are represented in structured expressions through some composition operators basically derived depending on task properties and temporal relationships specified in the task model. This phase also deals with appropriately translating transitions appearing within the PTS-based description onto appropriate interactors linking the various abstract presentations (we called them *connections*), so describing the dynamic behaviour of the system. In addition, some links with the functional core are identified. This step is referred by both (c1) and (c2) steps in Figure 22.

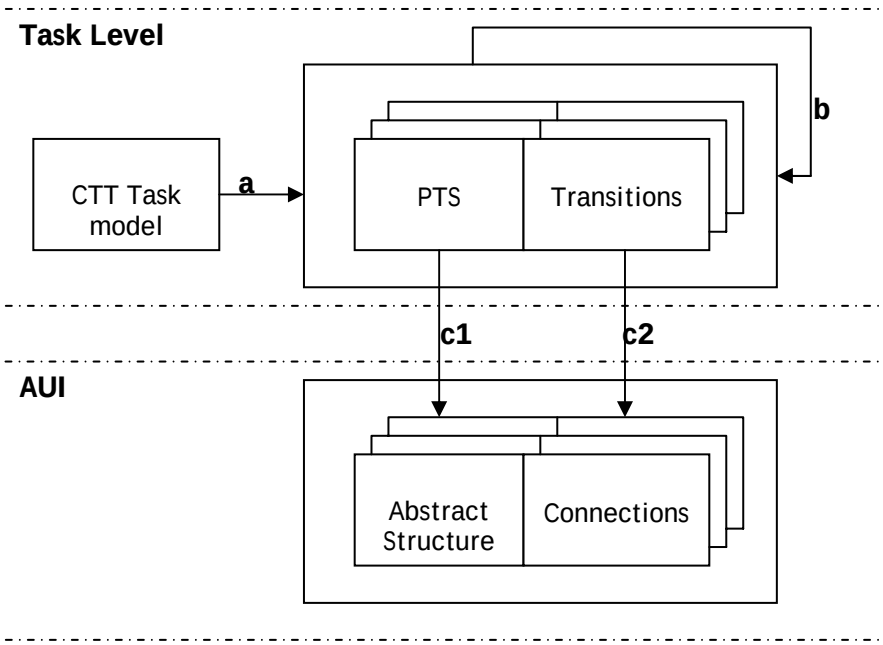

Figure 22: Transforming Task-Based Description into an Interactor-Based Description

Once we obtained the abstract description of the user interface, it should be concretised through a number of rules enabling the implementation of abstract objects through appropriate rendering techniques.

It is worth pointing out that some of the ideas that have been presented here have also been considered within a more articulated conceptual framework supported by a tool, TERESA, (http://giove.isti.cnr.it/teresa.html) that was developed having in mind multiplatform interactive applications. The tool was also useful to demonstrate the viability of the approach. The interest reader might refer to some publications (see for example Mori *et al.*, 2003; Mori *et al.*, 2004).

5.5. FROM TASK MODELS TO ABSTRACT USER INTERFACES

5.5.1. Identifying PTS and Transitions

The Presentation Task Sets (PTSs) are sets derived from a CTT task model, and represent tasks that are enabled over the same period of time. In particular, the PTSs are derived by analysing the formal semantics of the CTT temporal operators. For example, if two tasks are concurrently executed, they are enabled over the same period of time, so they should belong to the same Presentation Task Set. Alternatively, if two tasks are connected through an enabling operator, it means that the second task gets executed just after the first one ends up, so the tasks cannot belong to the same Presentation Task Set, as they will be enabled and executed in different periods of time.

The CTTE tool automatically calculates the list of Presentation Task Sets associated with a CTT task model according to a number of rules that have been identified for such a purpose. It is worth pointing out that such a calculation takes into account the temporal relationships specified within the task model, but expresses the calculated sets just in terms of the elementary interaction/application tasks appearing in the model itself. It is explained by the fact that only such tasks will have a 'physical' impact on the development of the user interface (differently for instance from pure cognitive activities), so they will be the only components appearing in the various PTSs. This is worth pointing out that, after this step what we obtain can be seen as just a different representation of the information contained in the CTT task model, still equivalent to the original one in terms of information provided and, consequently, at the same abstraction level.

As the reader should have realised at this time, the various PTSs allows for structuring the interactive system into a number of instantaneous, rough 'screenshots' of the panels in which the user interface will be finally articulated. However, the interactive system has a dynamic component that cannot be left out of consideration even when reasoning in terms of PTS. At this level, such dynamic behaviour is expressed by exploiting the conditions that allow passing from PTS to PTS, which depend again on the temporal operators existing among the various tasks. Such conditions can be presented in various manners. One example is when it is just a single task that allows enabling the next set of tasks (the simplest one is when one task is connected through an enabling operator with other tasks). So, in this case the condition coincides with the task itself. However, there are other cases in which such a condition reveals to be a Boolean expression involving two or more tasks. It happens when there is a group of tasks which are concurrently performed and this group is connected with an enabling operator to another set. In this case, whatever the execution order of those tasks, the condition for moving on to the next set of tasks is completion of all the concurrent tasks. As a result, the Boolean condition expressing such a constraint is an AND operator applied to all the involved tasks. In other cases the OR Boolean operator is necessary to express the fact that the performance of just one task in the set enables the next presentation. Moreover, also complex expressions with combinations of AND/OR operators can occur.

Within this section we have provided some basic rules for deriving PTSs in order to provide the users with the main concepts underneath them. A more complete description of the rules that have been identified for their derivation can be found in (Paternò *et al.*, 1998).

5.5.2. Applying Heuristics

As it has been explained before, the fact that two tasks belong to different Presentation Task Sets is connected to a sequencing specified in the task model between these two tasks, in the shape of an enabling operator. The direct consequence is that the number of Presentation Task Sets generated from the task model is of the same order as the number of the CTT enabling operators appearing in the task model. Therefore, a direct mapping between Presentation Task Sets and user interface presentations might produce excessively modal user interfaces or a high number of presentations with a very limited number of elements.

In order to cope with this problem, a number of heuristics have been identified for the purpose of helping designers to obtain a lower number of presentations by merging two or more PTSs into one set. The motivations for merging together two or more presentation task sets could be reducing the number of PTSs (which can be very high in some cases), or including within the same presentation significant information (as a data exchange is), even when the involved tasks belong to different PTSs, so that users can better follow the flow of information. Up till now, the heuristics that have been identified are the following ones:

- *Joining when Enabling*. If two (or more) PTSs differ for only one element, and those elements are at the same level connected with an enabling operator, they can be joined together.
- *Single Element Sets*. If a PTS is composed of just one element, it can be included within another superset containing its element.
- *Sharing Most Elements Sets*. If some PTSs share most elements, they can be unified in order not to duplicate information which is already available in another presentation in almost all parts. For example if the common elements all appear at the right of the disabling operator, they can be joined into one PTS.
- *Exchanging Information*. If there is an exchange of information between two tasks, they can be put in the same PTS in order to highlight such data transfer.

It is worth pointing out that it is the designers that decide about the heuristics' application, taking into account some parameters that might depend on the computing device on which the presentation will be rendered. This is a main issue especially for multiplatform applications: for example, if we consider graphical user interfaces, it is likely that on devices with small screens the heuristics will be less applied than on other devices with more extended capabilities, since e.g. on small displays too many user interface objects in the same presentation would tend to add clutter rather than increase usability.

5.5.3. Mapping PTS-Based Specifications onto Interactor-Based Descriptions

Once we have obtained the information about tasks belonging to each Presentation Task Set together with the transitions between them, the next phase is deriving the related abstract user interface, structured in terms of abstract presentations and connections among them within a structure that is illustrated in Figure 22. Hence, differently from before, now the nature of the elements composing our system changes in that we move from a task-based description to an interactor-based specification. A number of steps have been identified in order to perform such correspondence between classes of tasks with specific properties and appropriate interactor objects supporting their performance, more specifically, for every group of tasks identifying a PTS several steps are necessary, which will be further detailed in the following sections:

- Mapping basic tasks into abstract interaction objects;
- Identifying appropriate interactor composition operators;
- Mapping transitions into connections;

- Identifying links with the functional core.

The first two bullets mainly refer to the presentation part of an user interface, the third one concerns with the dynamic behaviour, whereas the last part deals with the identification of the links with the underneath functionalities of the application.

Mapping Basic Tasks into Abstract Interaction Objects

In order to map the various tasks into the related interactors, we have to consider task properties. First of all, the allocation of a task (whether the task is performed either through an interaction between the system and the user, or just by the application) is useful information to identify the category of the associated interactor (respectively, *interaction* or *only_output*). Each of these interactor categories represents a set of interactors identified by the type of the task supported. For example, an "edit" task type indicates that the corresponding interactor should allow editing of information, as well as a 'selection' task type indicates that the associated interaction techniques should support the performance of this kind of activity.

In Table 4 (resp.: Table 5) the information that has been taken into account to map interaction (resp.: application) tasks into the corresponding abstract interactors has been summarised. As we have explained in Chapter 3, for each CTT task category it is possible identify several task types, for instance, if an interaction task is considered, within this category a number of task types can be identified, specifying the particular intention that should be achieved through the performance of such tasks, which can be editing, selection, control, etc.. For each task type different parameters can be taken into account in order to identify the most suitable interactor to be used, because there is information that makes sense for a specific task type and does not make sense for another one. For instance, when we deal with edit tasks, we could wonder about the type of object manipulated by the task (is it a text? is it a numerical datum?), which could be useful to identify the best interactor able to support such an interaction. Alternatively, when we speak about selection tasks, we could be interested in the cardinality of the set from which the selection will be carried out.

Task Category = Interaction		***Interaction*** **Interactor**
Task type	**Condition**	
Edit	Class of Object =Text	Text_edit
Edit	Class of Object =Object	Object_edit
Edit	Class of Object =Number	Numerical_edit
Edit	Class of Object =Position	Position_edit
Control	Transition task	Navigator
Control	Non-Transition task, right part of a [>	Activator
Monitoring	Class of Object =Position	Object_edit
Responding to alerts	Class of Object =Object	Object_edit
Single Selection	Cardinality = Low	SingleChoice_low_card
Single Selection	Cardinality = Medium	SingleChoice_med_card
Single Selection	Cardinality = High	SingleChoice_high_card
Multiple Selection	Cardinality = Low	MultipleChoice_low_card
Multiple Selection	Cardinality = Medium	MultipleChoice_med_card
Multiple Selection	Cardinality = High	MultipleChoice_high_card

Table 4: Mapping *Interaction* Tasks onto Appropriate Abstract *Interaction* Objects

In order to express this information in Table 4 you can see that corresponding to the task of type "Edit" (first row of Table 4), depending on the class of object manipulated, the resulting interactor might change, ranging from an interactor allowing to perform just a textual editing, to the interactor allowing the user to edit a numerical datum, etc. Then, in order to understand the kind of interactor that we should use to map an edit task we should consider the type of object manipulated. As far as a control task is concerned, it is used in two possible situations: when the user wants to activate an internal functionality (activator) and when the goal is just to navigate within the different parts of the user interface. In this case the adopted rule depends not only on

the information specified within the task model, but also takes into account the PTS currently considered. In fact, if an interactor is associated with a transition, its goal will be to allow the user to navigate between different presentations, whereas, in the other case, an activator will be used expressing the fact that what is triggered is either an internal functionality, or the object allows the user to navigate within the same presentation, then no change of presentation is involved.

Task Category = Application		***Only_output*** **Interactor**
Task Type	**Class of Object**	
Comparison	Object	Object
Feedback	Any	Feedback
Overview	Object	Object
Overview	Text	Text
Overview	Description	Description
Locate	Object	Object
Grouping	Object	Object
Visualise	Object	Object
Visualise	Text	Text
Visualise	Description	Description
Generating_alert	Text	Feedback

Table 5: Mapping *Application* Tasks onto *Only_output* Interactors

The same type of rule is applied in the case of application tasks. 'Feedback' and 'Visualise' are examples of task types belonging to the application task category, they indicate the activity of presenting some results of a server-side application processing or some application data, so they will be mapped onto only_output interactors. Also in this case there are different interactors suitable to support these activities.

Identifying Appropriate Interactor Composition Operators

Within a presentation, when we have more than one interactor, we should identify appropriate operators to combine such objects. Several operators have been identified in order to appropriately compose in structured expressions the interactors belonging to the same abstract presentation. The interactor composition operators that appear in the abstract user interface are derived by analysing the CTT task model specification. In particular, not only are the CTT operators analysed, but also other attributes of the tasks (like the frequency) still take part in this transformation.

In fact, on the one hand, there are some CTT operators which are directly mapped onto corresponding operators of the abstract user interface. It is the case when two or more tasks sequentially performed end up in the same presentation task set (it is worth noting that it might occur only after applying some heuristics): the sequence at the task level is translated, at the abstract user interface level, through the application of the ordering operator.

Also, the relation operator mostly appears in the abstract description of the user interface when a disabling operator appears in the CTT task model. This is because the task at the right hand part of the disabling has an impact on all the tasks in the left side, then to emphasise this 1:N correspondence between the disabling task and the N activities that might be deactivated, this operator will appear in the abstract specification of the user interface. The grouping operator is the one with the least strict constraints (all the interactors belonging to a certain presentation might, in the final analysis, be grouped together)

Differently from the rules for the other abstract composition operators which mainly consider CTT temporal relationships, the application rule that involves the hierarchy operator strongly depends on the frequency values of the tasks involved: a high level of task frequency is indication that a task is recurrently performed, so the task might have greater 'importance' with

respect to other tasks that are less frequently performed, and the hierarchy operator was considered appropriate for conveying this kind of information.

Mapping Transitions into Connections

In this step the dynamic behaviour of the abstract user interface is obtained. More specifically, the different *transitions* connecting the PTSs at the task level will be mapped in appropriate connections (at the abstract level) defining the dialogue, through linking each other the various abstract presentations. This transformation is quite straightforward because the description of the PTS is structured in such a way that the part describing the dynamic evolution of the system is also expressed either in terms of simple transition tasks or complex expressions in which simple transitions are combined through Boolean operators. As you can note from the description of the abstract language in Section 5.3, also the connections might appear either as elementary connections or composite expressions where elementary connections are combined through Boolean operators. Therefore, the mapping limits to refer to interactors instead of tasks, and maintains the relationships expressed by Boolean operators, relationships that will be appropriately implemented in the more concrete level, through appropriate interaction techniques.

Identifying Links with the Functional Core

The last issue is how to connect the interactive part of a software application with the functional core, the set of application functionalities independent of the media and the interaction techniques used to interact with the user. Since in the task model the activities supposed to be performed by the system are identified by the application tasks, it is possible to identify in the task model the two situations that are relevant for such a link:

- when (the system tasks associated with) internal functionalities are supposed to perform information access to the back-end;
- when an internal functionality needs to present information to the user.

In order to manage these two cases, in each task specification the objects handled to perform the tasks are indicated, classified in *perceivable* and *application* objects: the first objects have a direct impact on the user interface inasmuch as they are associated with concrete interface elements appearing on the user interface (menus, buttons, images, labels, etc.), whereas the “application” objects refer to logical objects connected with the application underneath.

In the first situation no perceivable activity on the user interface side occurs, so it is possible to automatically identify the related tasks because they are system tasks characterised by a lack of perceivable objects in their specification. The interactors involved in such functionality are those handling the application objects whose values should be used to send a request to the functional core. In the abstract interactor there is information indicating what functionality of the core should be accessed (identified through the corresponding application task) and other attributes indicating the parameters associated to such a request. This information is further refined when moving to the concrete interface level.

Also the second case (when an internal functionality needs to present information to the user) can be automatically detected, because in this situation we have application tasks manipulating both perceivable and application objects. This means that the application functionality that has to present information to the user must communicate with the interactor handling the perceivable object associated with the task.

5.6. FROM THE ABSTRACT USER INTERFACE TO ITS IMPLEMENTATION

In the previous sections we illustrated how to use the information contained in task models to derive an abstract description of the user interface. Then, roughly speaking, we have understood how to start to generate the design of the interactive system in a way that should support as much as possible the logical information expressed in the model. The last step is now performing the mapping between the abstract level to the more concrete one. It will imply first of all mapping every abstract elementary interactor into a concrete object of the user interface and associate the abstract interactors linked to the transition tasks with concrete widgets able to appropriately implement the rules defining the dynamic behaviour of the user interface.

Moreover, also the abstract operators have to be appropriately implemented by highlighting their logical meaning: a typical example is the set of techniques for conveying grouping relationships in visual interfaces by using presentation patterns like proximity, similarity and continuity (Mullet and Sano, 1995).

5.7. A SIMPLE EXAMPLE

The left side of Figure 23 shows an excerpt of a task model while the right side -part (a)- shows the initial set of PTS automatically calculated by the tool taking into account the temporal relationships among tasks. As you can note, tasks that are on the right part of a disabling operator appear in the same set of tasks that they might disable, because, in order to be able to deactivate such tasks, they should be enabled in the same period of time.

In addition, it is possible to apply appropriate heuristics in order to combine two or more PTS together. For instance, in part (b) the result of applying the "*Exchanging Information*" criterion to PTS shown in part (a) is displayed. As you can note, PTS1 in part (b) derives from joining together PTS1 and PTS2 in part (a), as the two tasks "Show artist list" (system task of showing the list of available artists) and "Select artist" (the interaction task of selecting one of them) exchange information each other. In fact, while these two tasks have to be performed at different times (see the "[]>>" operator), they are so tightly connected that only one presentation is required to support both.

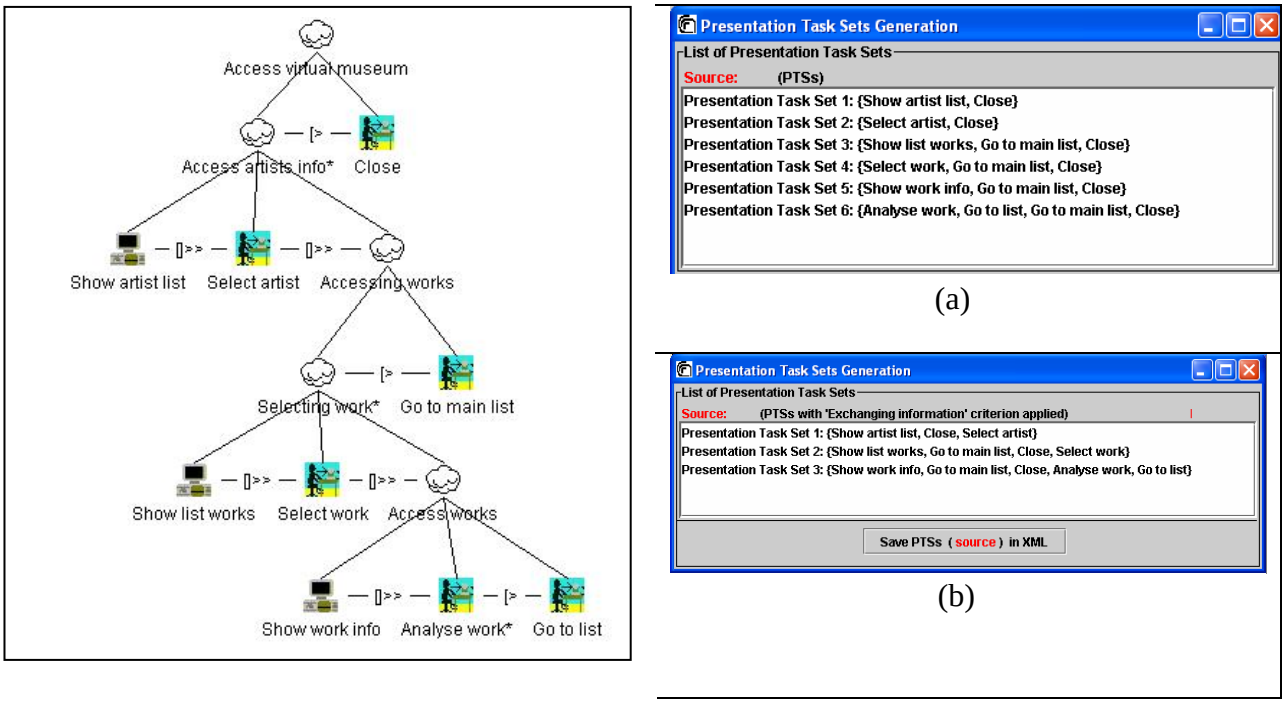

Figure 23: An Example of Task Model and related PTS Automatically Derived

Referring to the Presentation Task Sets obtained in part (b), we obtain three PTS that will be mapped into three corresponding abstract presentations. The problem is how to translate such a task-based specification into an interactor-based specification. Following the lines drawn out in Section 5.5, the steps that have to be carried out are first of all mapping basic tasks into abstract interaction objects, then identifying appropriate interactor composition operators and lastly mapping transitions into connections, as no link to the functional core exists in this case.

```
- <presentation name="presentation_1">
  - <connection presentation_name="presentation_2">
    - <conn_type>
        <elementary_conn interactor_id="Select_artist" />
      </conn_type>
    </connection>
  - <interactor_composition>
      <operator name="relation" />
    - <first_expression>
      - <interactor_composition>
          <operator name="ordering" />
        - <first_expression>
          - <interactor id="Show_artist_list">
            - <only_output category="onlyoutput">
                <text object="text" />
              </only_output>
            </interactor>
          </first_expression>
        - <first_expression>
          - <interactor id="Select_artist">
            - <interaction category="interaction">
              - <selection type="selection">
                - <single_choice type="single choice">
                    <singlechoice_medium_card type="medium card" />
                  </single_choice>
                </selection>
              </interaction>
            </interactor>
          </first_expression>
        </interactor_composition>
      </first_expression>
    - <second_expression>
      - <interactor id="Close">
        - <interaction category="interaction">
          - <control type="control">
              <navigator object="navigator" />
            </control>
          </interaction>
        </interactor>
      </second_expression>
    </interactor_composition>
  </presentation>
```

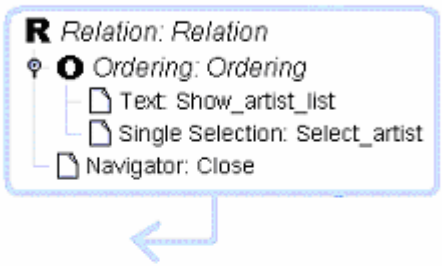

Figure 24: An Excerpt from an Abstract Presentation

We have that ‘Show artist list’ is mapped into a textual interactor visualising the list of artists, whereas the task allowing to select a specific artist is mapped onto a single_selection interactor with medium cardinality (the number of supposed artists), lastly the task that allows closing the presentation and moving to another presentation (‘Close’) is rendered as a navigator because it appears in the connection part of the presentation. As for the composition operators, we have that the presentation is structured (proceeding from the highest level to the lowest ones) into a relation expression in which the interactor associated to the change of presentation and enabling of the next one has an effect to both the interactors appearing in the nested ordering expression (this can mainly be referred back to the presence of a disabling operator in the task model). The motivation behind the ordering operator is in the fact that the related interactors are

connected with tasks that should be presented in different times but, as they are in the same presentation, this order is conveyed by the operator.

As for the implementation at the concrete level, we can say that, for example, the navigator interactor could be implemented through a button allowing the navigation between the two presentations, whereas for example the single_selection interactor associated with the selection of artists can be implemented through a pulldown menu. The fact that the “Close” button has an effect on all the other interactors of the presentation should be implemented by positioning the button in a suitable location within the window (for graphical user interfaces), in order to appropriately convey this information.

6. Using Task Models to Verify Properties of an Interactive System

Summary

In this chapter we present a method aiming to support the introduction of formal techniques in the design cycle of interactive systems and an environment supporting such a method. The main result is that designers can take advantage of the functionality of model-checking tools while they are still working with representations of the relevant models that are familiar with their practice. We explain how we use a CTT cooperative task model to reason about single and multi-user properties by accessing a model checker module through the representations provided by the tool for task model analysis and development. Thus, we show how tools can support such a method and what design choices have been made to improve the usability of such an environment, allowing even people with little background in formal methods to use it.

6.1. INTRODUCTION

In the last decade the computer science research community has put strong effort in developing tools and techniques for verifying requirements and design. The most successful approach that has emerged is *model checking*, whose essential idea is shown in Figure 25. A model-checking tool accepts a model specifying the user requirements and some properties that the final system is expected to satisfy. The tool then yields a positive answer if the given model verifies the property, otherwise generates a counterexample detailing why the model does not satisfy the specification. By studying the counterexample, it is possible to pinpoint the source of the error in the model, correct the model and try again. The main idea is that by ensuring that the model satisfies a reasonable number of system properties, we increase our confidence in the correctness of the model itself.

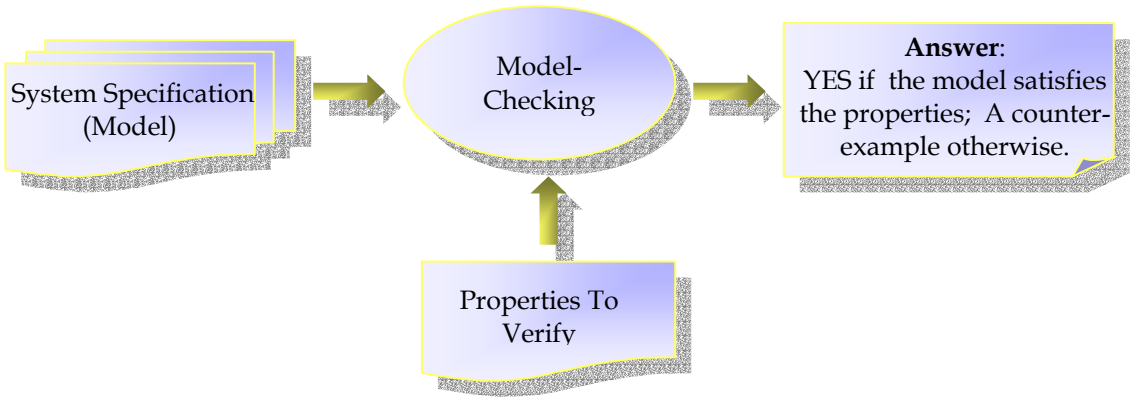

Figure 25. The Model Checking Approach

There are various motivations to carry out model checking for user interfaces. Firstly, it is possible to reason about an application also before it has been developed (or when it has been implemented only partially). To this end, the model should be a meaningful approximation of the application in order to support a meaningful analysis. Secondly, user testing can be expensive, particularly in highly specialised fields such as ATC, where the users is personnel whose time has high costs. In addition, it can be very difficult to determine how many tests are needed to obtain an exhaustive analysis. Model checking can decrease the problems found at the stage of empirical testing, so reducing expensive changes late in the design process. Finally, model checking allows an exhaustive analysis, whereas in user testing we just consider one of the possible sequences of actions, whereas a huge number of such traces may exist and even extensive empirical testing can miss some of them, and this lack of completeness in empirical testing can have dangerous effects especially in safety-critical contexts.

In hardware design, where it is important to check whether some properties are satisfied before implementing the specification, formal verification has been successfully and widely used. In particular, model checking has proven to be a tremendously successful technology to verify requirements for real-time embedded and safety-critical systems. The HCI field is more challenging for verification methods and tools, since the specification of human-computer dialogues might be more complex than the hardware specifications as the human behaviour can be affected by a plethora of aspects.

As a matter of fact, formal methods have stimulated limited interest in the HCI area especially in industrial environments because it has been traditionally claimed that dealing with them is too difficult, because even the related tools offer insufficient support to allow their easy and effective use. Actually, since formal approaches require formally describing an interactive system and identifying significant properties, in order to use them it is generally also required to know how to formally express these properties and interact with the underlying tools: for most user interface designers and developers this effort is enough to discourage them from adopting such approaches. Therefore, the problem of how to facilitate the acceptance of such techniques is multi-faceted.

6.2. RELATED WORK

The importance of formal specification for the design and implementation of interactive applications has already been pointed out. Early work in the area of formal methods for HCI mainly aimed to formally express relevant concepts (Dix, 1991) but afterwards interest in a more systematic introduction of formal techniques in the design cycle has arisen. To this aim, different approaches can be used to verify properties on interactive systems (model checkers, theorem provers) with each approach able to better address specific issues and suit different verification needs. In particular, model-checking techniques have been seen as a useful support for a more systematic analysis of relevant models (Abowd *et al.*, 1995; Campos and Harrison, 2001; Palanque and Bastide, 1990; Paternò and Mezzanotte, 1995; Rushby, 1999). A kind of model checking approach is also the work by Cairns and others (Cairns *et al.*, 2001) where Markov models are analysed through "knowledge/usability graphs" even if in this approach the demonstration of formal properties is not considered.

Formal models are notoriously hard to build and their value has often been criticized, mainly because of their limited usefulness compared to the required effort. In addition, for usability evaluation, there is a lack of well defined properties that relate to usability (vanWelie *et al.*, 1999b) and they are usually limited to dialog aspects and they ignore presentational aspects. A few works have addressed the issue of supporting designers while formalising properties. For example, in (Lusini and Vicario, 1998) a visual formalism for the presentation of a real-time logic is proposed, based on a visual metaphor allowing mapping textual sentences onto visual representations. The resulting notation is supported by an interactive syntax-driven editor integrating the visual presentation with the conventional textual notation. Also in (Del Bimbo and Vicario, 1999) a visual system is described which supports visual presentation and manipulation of temporal Computation Tree Logic (CTL) (Clarke *et al.*, 1986) properties by means of metaphoric transposition of the abstract temporal semantics of the logic onto a concrete visual space: the use of 3D graphics, multiple windowing, colour associations and the exploitation of both textual and graphical representations allow obtaining a complex interaction environment, which also supports visual editing of temporal expressions.

Regarding the integration between formal methods and user interface design, in recent years interest in such tools has grown with the aim to improve the reliability of the resulting software. However, they may result difficult to manage. In order to ease their application, a number of tools have been proposed. For example, the Preventing User Errors project (Curzon

and Blandford, 2001) regarded the development of tools and techniques to detect design flaws in interactive systems and to prevent user errors by integrating the formal user modelling with formal system verification (in this project, the HOL theorem prover is used). However, as it has been claimed in (Campos and Harrison, 2000), given the richness, complexity and number of different possible perspectives on interactions, trying to adopt a unified and monolithic specification will result in a very complex model, too complex for verification. Thus, instead of having a single global model, a number of partial, property-oriented specifications of the system should be built, allowing selection of the most appropriate technique in each specific case, because different types of properties require different proof techniques and different specification styles. This type of approach has several advantages and also some problems: for example the number of restrictions imposed on the notation used limits the expressiveness of the approach and, at the methodological level, the use of multiple specification techniques inevitably raises the issue of consistency between the models produced with each technique.

Another attempt to use model checking techniques in this area was the work presented in (Loer and Harrison, 2000), which involves specification of the properties to be checked, verifying those properties and analysing the traces produced by the model-checking tool if a property does not hold. However, in contrast to the main attention of this approach to functional aspects and system-centred issues, our method uses task descriptions whose elements can capture and give meaning to checkable properties, while still using model-checking techniques. One traditional factor limiting the acceptance of tools for formal methods is that they are hard to use. We need more usable environments for many purposes: representing the model of interest, specifying properties, interacting with the underlying model-checking tools, and presenting results provided by the model-checker. In addition, applying model checking techniques to the design of user interfaces poses further problems:

- *The identification of relevant user interface properties to be checked*: the properties should be general enough to fit different application domains and, at the same time should be able to take into account some specific requirements of the application domain considered.
- *The formalisation of those properties*; this is the difficult part of the process and the one which discourages many designers when using this approach. There are few tools that ease the use of languages to specify properties, and even when people know a given language, additional constraints might force them to learn other languages in order to use a specific model-checker that supports the desired functionality.
- *The development of a model of the User Interface System* which has to be meaningful and, at the same time, avoids the introduction of many low level details which would increase the complexity of the model without adding important information for the design of the user interface.

In the next section we present our perspective about introducing formal techniques in the design cycle of interactive systems. The expected objective is enabling designers to take advantage of the functionality of model-checking tools while they are still working with representations of the relevant models that are familiar with their practice. In this way, even people with little background in formal methods should be able to benefit from such approaches and easily use them.

6.3. THE APPROACH

As it has been described in Chapter 5, an interactive application is characterised, from a presentational point of view by the dialogues it supports and the presentations of information

that it generates for communicating information to the user. The description of both these aspects could be formally represented. However, the introduction of a formal representation has to be motivated. For instance, the effort to formalise presentation aspects does not seem to be justified because in such a way we would obtain models describing features that might be easily understood by direct inspection of the implemented user interface, as they are strictly related to how people perceive information. Instead, the case of user interface dialogues and the possible sequencing of user and system actions at the interface level is different. In this case, there are aspects that are more difficult to grasp with an empirical analysis because when users navigate in an application they follow only one of the possible paths of actions.

However, the separation of presentation from dialogue poses several problems and there are cases where important user interface issues involve both lexical/presentation and dialogue issues (Dix, Finlay, Abowd, Beale, 1998). In addition, interactive systems are highly concurrent systems because they can support the use of multiple interaction devices, can be connected to multiple systems, can support the performance of multiple tasks, and often can support multi-user interactions. The current tendency does not ameliorate the situation as it rather tends to increase even more such concurrency as it is a source of flexibility, usability, and interactive richness but, at the same time, it generates complexity that needs to be carefully considered, especially in safety-critical contexts. Thus, formal techniques can provide useful support to better understand dialogue models and their properties.

In addition, as a designer needs first to understand the logical and temporal relationships among the logical activities (which to some extent depend on the artefacts available) a representation supporting such information is required. Formalising task models can allow designers to reach a number of results: a better understanding of logical activities to be supported; obtaining information useful for the concrete design of the user interface, supporting usability evaluation of the application considered, rigorously reasoning about properties. The last aspect has not been considered sufficiently and will be addressed in this chapter on the assumption that *the dialogue aspects should be taken into account when formalising HCI aspects with particular attention to their relationships to tasks and their performance.*

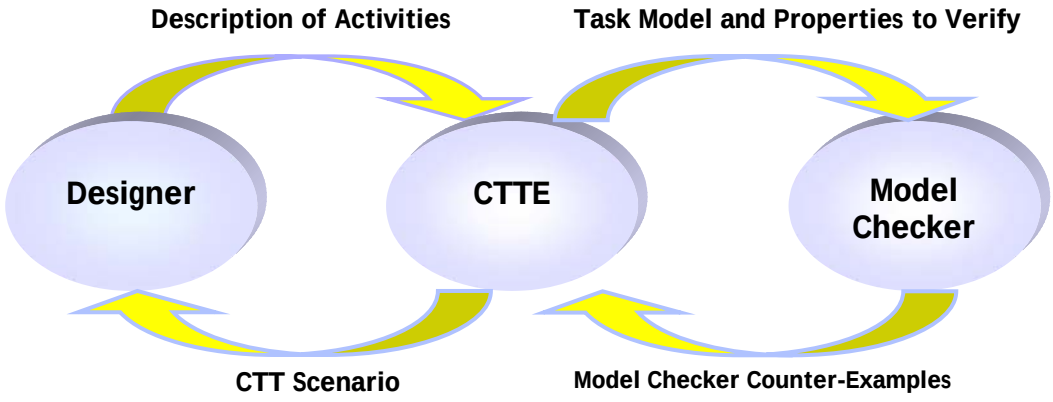

Figure 26. Input and Output Flows between CTTE and Model-Checker

To summarise, the main objective of our approach (see Figure 26) is to support model-based design through a set of tools that allow designers to easily move from the informal descriptions of both the interactive system and its related properties to their respective formal specifications. We are able to reflect back the model checker results in the language of the original specification, thus overcoming one limitation of many proof checkers, which have been difficult to be used because they are not able to reflect back problems in the proof in an intelligible form. By using an extension of the CTT Environment (Mori *et al.*, 2002), the designer is able to obtain the formal specification of activities supported by the system, together with the formal properties to be verified, which both represent the main input of the model-

checking tool. If the property is not verified, the model-checker returns as output a possible counter-example that is mapped into a specific sequence of tasks (CTT scenario) that the designer can further analyse and interactively simulate within CTTE (see Section 3.3.2 for simulator details). It is worth noting that in this way the use of the model-checker becomes completely transparent to the users. Whatever the specification language of the model checker, the users specify both the interactive system and the properties in terms of the tasks involved; whatever the kind of low-level actions the model checker returns as a result of the verification phase, the designers will be provided with task-related information.

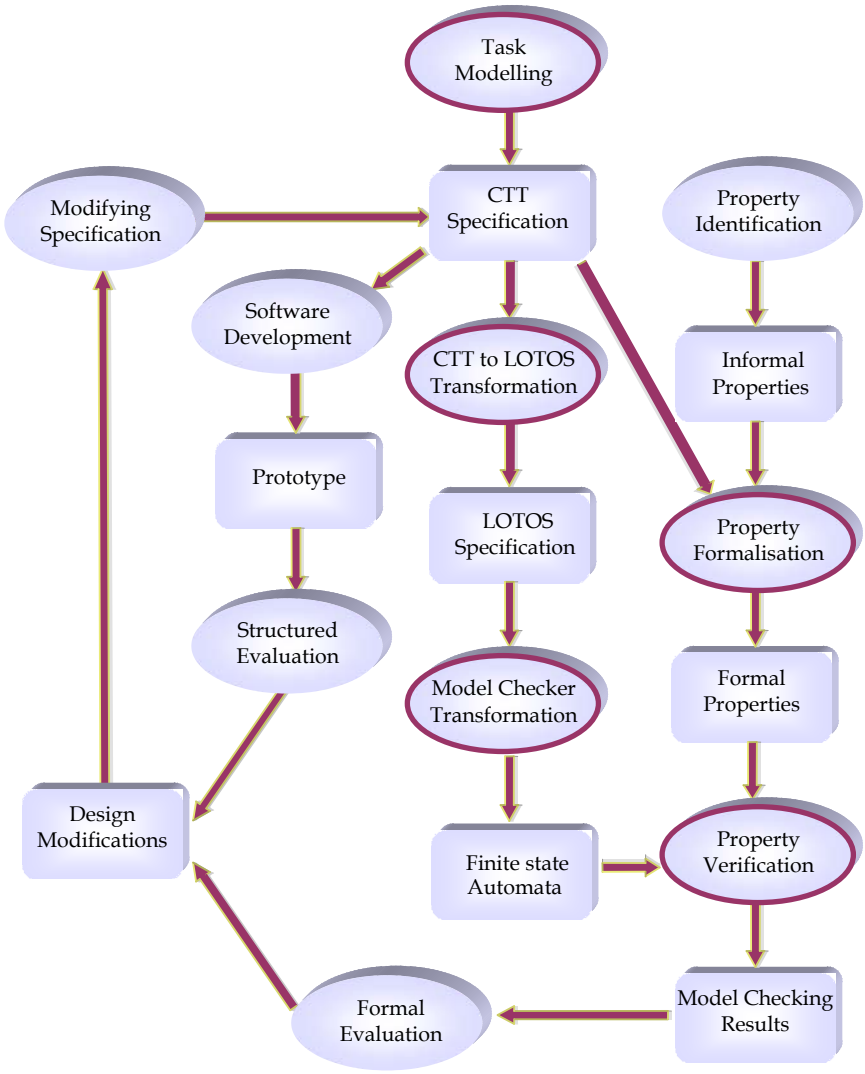

Figure 27. An Approach for Integrating Task Modelling and Model Checking

The method that we propose is represented in Figure 27, where the processes are indicated with ovals and the results with rectangles. Some processes are tool-supported and in Figure 27 we highlighted them by means of a stronger border, namely the phases concerning task modelling, the CTT-to-LOTOS transformation and property formalisation, which are all are included within an extension of the CTTE tool that will be presented in the next sections. All the other tool-supported phases (model-checker transformation and property verification) are provided by a model-checker included in the CADP environment (Garavel *et al.*, 2001). More precisely, our approach comprises the following activities:

- *Task Modelling*. After a first phase gathering information on the application domain, and an informal task analysis, designers should obtain a task specification by first structuring the tasks in

a hierarchical way and then defining the concerned temporal relationships. Using the CTT notation, we obtain a specification of the cooperative application with one task model for each role involved and one part specifying the relationships among tasks performed by users with different roles (see Section 3.2.3 for further details about specifying cooperative applications in CTT) thus giving a clear indication of how cooperations are performed.

- *Software Development.* The task model can be used to drive the development of a software prototype consistent with the indicated requirements. This problem is directly connected with the approach described in Chapter 5, where we have shown an approach in which at first a logical, abstract description of the user interface is derived, and then it is mapped onto a concrete one.
- *CTT-to-LOTOS Transformation*. The other possibility is transforming the CTT model into a LOTOS specification to analyse potential task performance and the user interface dialogues by using model checking techniques, by exploiting the fact that CTT temporal operators extend the basic set of LOTOS operators. We implemented a transformation tool where each task specification is translated into a corresponding LOTOS process. The reasons for it is that there are various model-checking tools able to accept LOTOS specifications as input (such as the CADP package that we used).
- *Property Formalisation*. The identification of the relevant properties of the user interface considers both general HCI properties, such as the continuous feedback property, and other properties specific to the considered application domain. To support user interface designers while editing formal properties, we have defined a set of templates associated with relevant properties so that the designer has only to fill some parameters for identifying the tasks involved in the property. Such tasks are directly selected on the graphical representation of the task model.
- *Property Verification***.** After having formalised the identified properties with a formal notation, the next step consists of applying the Caesar compiler (Garavel *et al.*, 2001) which translates the behavioural part of a LOTOS specification into a Labelled Transition System (LTS). In our integrated environment the property verification phase is performed by the EVALUATOR model-checker (Mateescu, 1998), and its results are used for formal evaluation: for example, in case of negative results, the counter-example provided by the model-checker is mapped into a task sequence, thus allowing an easier and more meaningful analysis.
- *Modifying Specification.* Both the informal evaluation of the prototype and the results of model-checking can provide useful suggestions for improvements that can be used to update first the task model and the corresponding prototype and model for automatic verification. After such updating phase, the overall process can iteratively start again from the beginning.

6.4. FROM CTT TO LOTOS

In order to perform model checking of CTT task models we translate such models into LOTOS. The translation has first to identify what the correspondent of tasks, the elementary "bricks" of a CTT task model, are in LOTOS. A CTT task is an activity that can be possibly decomposed in a hierarchical way into smaller, concurrent sub-tasks. Likewise, in LOTOS a concurrent system is seen as a process, possibly consisting of several, concurrent sub-processes. Then, we have that CTT tasks and LOTOS processes are similar in that both are concurrent entities that can be decomposed, then a *CTT task translates into a LOTOS process*.

```
specification typical_spec [ gate list ] ( parameter list ) : functionality
type definitions
behaviour
behaviour expression
where
type definitions
process definitions
endspec


process definitions:

process typical_proc [ gate list ] ( parameter list ) : functionality :=
behaviour expression
where
type definitions
process definitions
endproc
```

Figure 28. Typical Structures of Specification and Process Definitions in LOTOS

In Figure 28 the structure of a typical LOTOS specification is described. Apart from *type definitions* (e.g. Boolean, Natural, ..) and *parameter list* (which is a list of variable declarations) both included in full LOTOS specifications but not considered in our approach, the description of a LOTOS specification is composed of three main components:

- behaviour expressions
- process definitions
- functionality

Now we describe the process definitions of a general leaf task, then the process definition of a general high level task. Next, we start to consider the unary operators of CTT to be translated into LOTOS, namely the iterative and the optional operators. For each operator we consider the two cases of a leaf task and a high level task. Finally, we move on considering the other binary operators which have to be translated into LOTOS (order independency and suspend-resume), by providing their behaviour expressions. In the following paragraphs we denote, for each process P, *gates(P)* the list of elementary gates appearing in P, with *Process_definition(P)* the definition of process P, and *functionality(T)* the way in which the activity terminates.

The most elementary CTT task structure is represented by a leaf task, which describes the performance of just one elementary action. The correspondent of this structure in LOTOS is a process whose behaviour is represented by just this action offered at its unique interaction point or gate, and a keyword denoting how the process terminates, with or without success (*exit* or *noexit* respectively). It is worth noting that we have to calculate the kind of termination of a LOTOS process depending on the structure of its behaviour expression, whereas in CTT it does not have to be explicitly defined.

6.4.1. Single-User Task Models

In this section we consider the translation from CTT to LOTOS when single-user task models are considered. We provide a number of rules that will be applied depending on the structure of the considered tasks, starting with a CTT leaf task not iterative as highlighted by the following rule:

Rule 1: T is a CTT *leaf* task, not iterative

process T_proc[T]: *exit* :=

T; *exit*

endproc

If we consider a high level CTT task, its specification is just the composition of lower-level tasks; in the same way, the behaviour expression of a complex LOTOS process is defined in terms of the behaviour expressions of its sub-processes. As a result of such decompositions, in the same way in which in CTT the observable actions of a high level task reduce just to leaf tasks, in the LOTOS process definition of a complex process all the elementary actions appearing in its sub-processes will be listed:

Rule 2: T is a *high level* CTT task, T= t_1 op_1 t_2 … op_m t_{m+1} with t_1, t_2 … , t_{m+1} children of T

process T_proc[*gates*(T)] : *functionality*(T) :=

t_1_proc [*gates*(t_1)] op_1 … op_m t_{m+1}_proc [*gates*(t_{m+1})]

where

< *Process_definition*(t_1)>

…

< *Process_definition*(t_{m+1})>

endproc

Once the CTT tasks are mapped onto LOTOS processes we have to define how to express the CTT temporal operators in terms of LOTOS expressions. Regarding the translation of the CTT operators, some of them directly derive from LOTOS (this is the case of the *choice* operator [], the *deactivation* operator [>, the *independent concurrency* operator |||, the *recursion* operator). Other CTT operators are extensions of the previous ones, obtained by highlighting the possibility of exchanging information over a task synchronisation (the data exchanged are the values offered at the gates of the processes involved). This is the case of the *concurrency with information exchange* operator (|[]|) and the *enabling with information passing* operator []>>. Other operators need to be mapped into new LOTOS behaviour expressions, as they do not have any direct correspondent in this formalism. These operators concern optional tasks, iterative tasks, the order-independence operator, and the suspend-resume operator.

If a task T is optional its behaviour is expressed by means of a LOTOS process decomposed into two sub-tasks: a no-action process and the process itself connected through a choice operator with the first one. Therefore, the translation rule becomes the following one:

Rule 3: T is a CTT *optional* task ([T] in the CTT notation)

Behaviour_Expression([T]) = T_proc[*gates*(T)] [] *exit*

As far as the implementation of an iterative task is concerned, its behaviour requires a recursive call of the LOTOS process associated with the task, in order to simulate the behaviour

of restarting an activity just after the completion of the previous execution. For the iteration we distinguish between two cases, depending if the task is a leaf or not. In case T is a basic task then T* translates according the following rule:

Rule 4a: T is a CTT iterative (T* in the CTT notation), *leaf* task

process T_proc[T]: *noexit* :=

T; T_proc[T]

endproc

If T is an iterative high level task, the translation of the task will be expressed in terms of the related expressions associated to its subtasks. The associated rule is described in the sequel:

Rule 4b: T is a CTT iterative (T* in the CTT notation), *high level* task, T= t_1 op_1 t_2 … op_m t_{m+1} with t_1, t_2 … , t_{m+1} subtasks of T

process T_proc[*gates*(T)]: **noexit** :=

(t_1_proc [*gates*(t_1)] op_1 **…** op_m t_{m+1}_proc [*gates*(t_{m+1})])

>>

T_proc[*gates*(T)]

where

< *Process_definition*(t_1)>

…

< *Process_definition*(t_{m+1})>

endproc

The other case to consider concerns the order-independence operator, which basically means that the two involved activities can be performed regardless any specific order in the sequence. More specifically, if two tasks A and B are connected with this operator, this behaviour is equivalent to the LOTOS behaviour expression ((A>>B) [] (B>>A)). If we have more than two tasks, the reasoning can be easily generalized to express the possibility of performing the various tasks in any sequence.

Rule 5: T = T1 |=| T2 (CTT Order-independence operator)

Behaviour_Expression(T1 |=| T2) =
(T1_proc[gates(T1)] >> T2_proc[gates(T2)])
[ ]
(T2_proc[*gates*(T2)] >> T1_proc[*gates*(T1)])

The aim of the suspend-resume operator |> is to specify the ability of a task to suspend temporarily and then resume, a situation that is very commonly found in real applications. If we have A|>B, this operator specifies that B is a suspending behaviour with regard to A, that is, B can pause A in order to be performed and after its completion A will resume at the point of its interruption by B. Unfortunately, the suspend-resume operator proves to be quite cumbersome to specify with standard LOTOS operators. For this purpose, we designate as a_i [i= 1, …,n]. each basic task derived from the decomposition of A. By basic tasks, we mean tasks that are

elementary and so cannot be further decomposed into subtasks. In order to express the ability of B to repeatedly interrupt A, the behaviour (A|>B) is equivalent to a new expression obtained by simply replacing each a_i [i= 1, …,n] of A with another expression which is the choice between a_i and B, with B followed by the recursive instantiation of such choice. This means that if a_i is performed, we can carry out the next basic task of A (a_{i+1}), otherwise B is performed and then it is still possible to choose one of the two possibilities.

Rule 6: T = T1 |> T2 (CTT Suspend-resume operator). If T is a *high level* task, T= t_1 op_1 t_2 … op_{n-1} t_n with t_1, t_2 … , t_n subtasks of T

Let C= (t_i [] D) where D= T2>>C

Replace every occurrence of t_i with C expression.

6.4.2. Cooperative Task Models

In the previous section we defined the rules that allow mapping each CTT task into a LOTOS process and in this way we can obtain the definition of the LOTOS process associated to a single task model. However, it is necessary to take a step further to complete this transformation if we consider cooperative CTT task models. In fact, with such task models we have to combine the process definitions of each role with the process definition of the cooperative part by means of appropriate operators in order to obtain the correct translation of the original cooperative CTT task model specified.

Consider a CTT specification of a cooperative task model constituted by (Role_1, Role_2, …, Role_n, Cooperative) task model. With cooperative task models we have a number of task models associated with the different roles (Role_1, … Role_n) and one cooperative task model. Thus, the translation into LOTOS of such a model not only has to provide the specifications associated with each role and with the cooperative part of the model, but also the description of the inter-process relationships amongst them.

With the rules previously explained we easily obtain the specification of the process associated with each role and the specification associated with the cooperative part of the task model. For each i=1, …,n we indicate such descriptions respectively with *Process_definition(Role_i)* and *Process_definition(Coop_part)*. What is still missing is to obtain the specification of the behaviour expression describing the relationships amongst all those processes. Such relationships will specify basically that each part of the cooperative model performs its own activities in a pure interleaving with the activities of the other components. However, synchronisation will be required on those tasks which take part in cooperative activities (namely, such tasks correspond to ‘connection’ tasks).

This overall behaviour is easily described in LOTOS, as such a language provides the *parallel composition* operator (|[g1,…, gn]|), which allows for describing exactly such kind of process synchronisation over a specified set of actions (g1, …, gn). The cooperative task model we considered will correspond to the specification labelled as (1).

The only part that have still to be described is the behaviour expression associated with the overall task model. As we said before it corresponds to the process associated with the cooperative part (*Cooperative_Root_proc*) which synchronise over the connection tasks with the behaviour expression obtained by concurrently performing the processes associated with the roles (Role_1_proc, …, Role_n_proc). So, the *behaviour expression* of (1) corresponds to that specified in the shadow box labelled (2).

Rule 7: T is a CTT cooperative task model

specification Coop_spec [*gate list*] : *functionality*
behaviour
behaviour expression (1)
where
process definitions
endspec

under the following conventions:

- *gate list* is the list of all the gates (or elementary actions) appearing in the cooperative task model; If we denote with gates(P) the list of gates of process P, *gate list= gates(Role_1) U … U gates(Role_n).* It is worth noting that it is not necessary to add the list of gates for the cooperative part, as such gates already appear within the gates of the roles to which they belong.
- *functionality* is the calculated functionality of the cooperative task model calculated according to a set of defined rules;
- *process definitions* is the set of the definitions of all the processes appearing in the specifications, namely: Process_definition(Role_i), … Process_definition(Role_n), ,Process_definition(Coop_part).

Behaviour expression of (1) :

```
Cooperative_Root_proc[gates(Coop)]
 |[gates(Coop)]|
 (                                                    (2)
  Role_1_proc[gates (Role_1)]
   |||
   Role_2_proc[gates (Role_2)]
   |||
   …
   Role_n_proc[gates (Role_n)]
  )
```

under the following conventions:

- *gates (Coop)* are the elementary actions of the Cooperative part (namely the CTT "connection" tasks)
- for each i=1, …, n *gates (Role_i)* are the gates of each Role_i.

To summarise, in this section we have identified a number of rules that allow translating CTT task specifications into LOTOS processes. In the next section we will describe and formalise some sample properties that we identified relevant to be checked.

6.5. PROPERTY FORMALISATION AND VERIFICATION

6.5.1. The Approach

The properties have to be formalised in a language that can be accepted by the model-checker installed in the adopted tool. EVALUATOR is the model-checker included in the CADP package, used in our approach for the property verification phase.

The temporal logic used as input language for the EVALUATOR is the regular alternation-free mu-calculus, an extension of the alternation-free fragment of the modal mu-calculus (MCL) (Emerson and Lei, 1986). with action predicates and regular expressions over action sequences. The version of EVALUATOR that we used for our integrated environment handles a version of the logic that lacks support for data values.

The properties should allow the designer to handle the aspects that they have considered relevant to their system, because such properties do depend on the system under consideration. Examples of properties considered are reported in the following Section 6.5.2.

The verification phase yields as a result a Boolean value displayed on the standard output, possibly accompanied by a *diagnostic* explaining the truth value of the formula. Usually, a diagnostic is a (generally small) portion of the graph where the property yields the same result as if it is verified on the whole graph. It can be a sequence of actions but it can also be more complex (e.g. a graph with cycles). Diagnostics can be generated both when the property verification returns true and when the property verification returns false. Far more significant is the case when the verification returns false, because in this case the diagnostic includes the set of counter-examples that make the property false, showing one (or more than one) possible execution path that does not satisfy the property. In this set there could be more than one element, whereas designers often want just *one* counter-example to examine. In order to extract a specific sequence of actions from the diagnostic, a random execution on the graph corresponding to the diagnostic can be carried out. The analysis of such cases can be an aid to discover points in the specification where the behaviour has to be modified, or another way to check that some conditions are verified in specific cases. If the counter-example provided is considered not particularly relevant, then it is possible to ask for additional ones.

6.5.2. Sample Properties

In this section a small number of properties will be described. As it has been previously pointed out, the properties should allow the designer to handle the aspects that they have considered relevant to their system, because such properties depend on the system under consideration. For example in safety-critical systems as the ATC domain we analysed, we might consider important all the properties that have an impact on the safety aspects of the system as e.g. cooperative aspects regarding communication between different users are. For each property a formal specification is provided (in a language accepted by the model-checker), together with an informal explanation of its meaning. All the templates have been written by exploiting a library allowing designers to use the more concise syntax of ACTL (De Nicola *et al*., 1993) operators that are mapped onto MCL constructs.

Permanent Reachability

With this template we want to verify if in any state of the system (G operator) for all the possible temporal evolutions (A operator) it is possible to execute a certain task. The formal template provided is

```
AG (< task > true)
```

where *task* is the variable that has to be instantiated before the property is passed to the model checker. For example, the designers would verify that the controller is able to use the special channel reserved for communicating urgent clearances to aircraft anytime.

Absolute Reachability

The goal of this template is checking whether in any state of the system a temporal evolution (E operator) during which it is possible to reach a state (F operator) where *task* can be executed exists. The corresponding formal template is the following:

```
AGEF (< task > true)
```

An example of application of this task concerns the possibility for the controller to communicate via phone with another controller in a different sector.

Existential Relative Reachability

In this case when we want to check if, after having performed *task1*, at least one possible temporal evolution exists in which it is possible to perform (U operator) *task2*.

```
AG [task1] E[ true {true} U {task2} true ]
```

The various occurrences of "true" in the above expression indicate that there is no restriction in terms of actions to perform during the transitions and in terms of states. In the considered case study, an example is the possibility for the ground controller to ***stop an aircraft*** (**task2** in this case) after having previously allowed it *to* ***follow a taxiway*** (**task1** in the template).

Universal Relative Reachability

In this case we want to check if, after having executed *task1*, in all the next temporal evolutions of the system it is possible to execute *task2*. This property is stronger than before because we consider all temporal evolutions instead of at least one possible (*A* operator instead of *E*). The formal template of this property is the following:

```
AG [task1] A[ true {true} U {task2} true ]
```

As an application example of this property we could instantiate *task1* and *task2* with the same tasks examined in the previous property example ("follow a path" and "stop an aircraft" respectively for task1 and task2). However, differently from the *existential* relative reachability, now we are considering the *universal* relative reachability property, which is stronger, and in fact, the *universal* relative reachability instantiated with those two tasks is not verified in the system. If it had been verified, we would have required that *whenever* the pilot receives the order of *following a specific taxiway* (*task1)*, the controller inevitably will stop him/her (*task2*) during his path, which is evidently not true (the pilot is stopped by the controller only when abnormal situations occur).

Property	ACTL Expression	Explanation
Permanent_Reachability(task)	AG (<task> true)	In any state of the system it is always possible to execute task
Absolute_Reachability(task)	AGEF (<task> true)	In any state of the system it is always possible to reach a state where it is possible to execute task
Existential_Relative Reachability(task1, task2)	AG [task1]E[true{true}U{task2}true]	In any state of the system it is possible to execute task2 after having executed task1
Universal_ Relative Reachability(task1, task2)	AG[task1]A[true{true}U{task2}true]	After having executed task1, in all the next temporal evolutions task2 will be performed

Table 6. Examples of Property Templates Identified

In Table 6 a summary of the property templates that have been identified is shown: in the first column the name of property appears, in the second one the related formal ACTL expression is reported, whereas in the last column an explanation of the property is provided. In the next section we will describe the different features that the integrated environment we developed offers for the design of interactive systems with the support of formal methods

6.6. INTEGRATION OF TOOLS FOR TASK MODELLING AND TOOLS FOR MODEL-CHECKING

The main phases of this framework are currently supported thanks to an automatic environment allowing an integrated use of two tools, CTTE, and a model-checking included in the CADP package (although our environment can be easily integrated with other similar model checking tools). Within such environment, designers can edit a task model and directly access to some functionality of the underlying model checking tool (*Activate Model-Checking Tools* item in the *Tools* menu, see Figure 29).

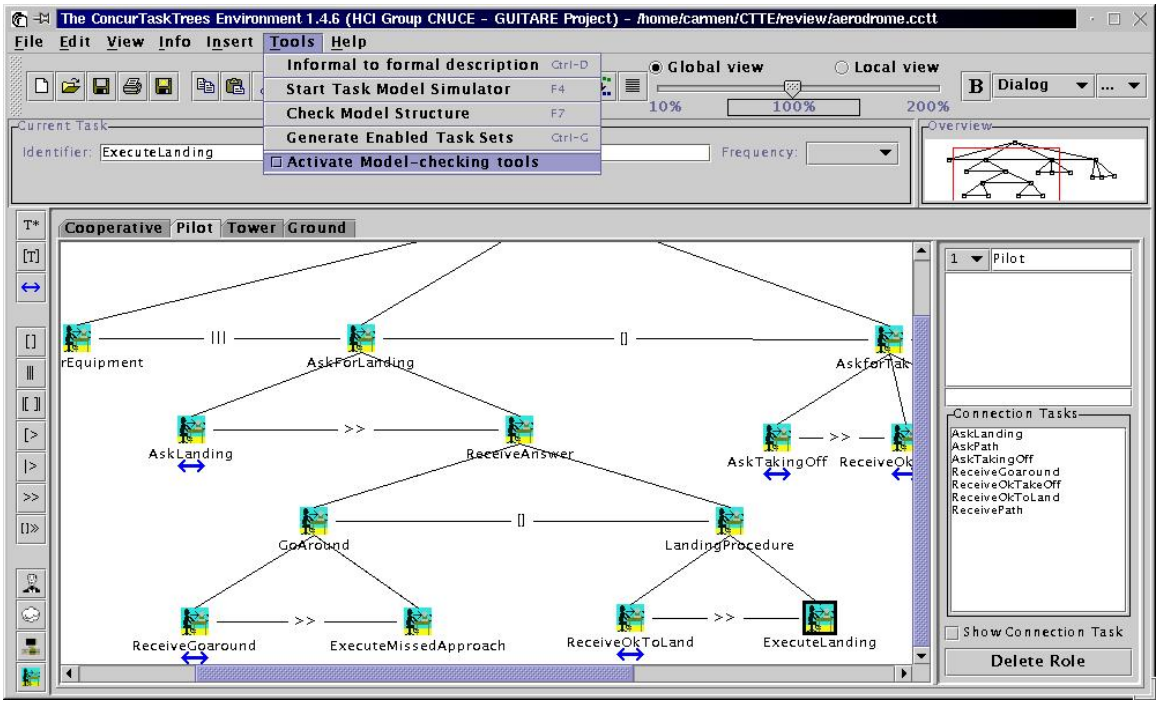

Figure 29. Activation of the Model-Checking Tools

Once such a tool is activated, a new window appears showing all the provided functionality divided into two main panels (*Commands for LOTOS specification* and *Commands for LTS automaton*, see Figure 30). Depending on the particular panel that has been selected, the window shows different sets of commands. One panel includes commands used to get and handle LOTOS specification, e.g. generate LOTOS code (*From CTT to LOTOS* button), indent it (*Indent LOTOS spec*), produce the Abstract Data Types for it (*Translate ATD*).

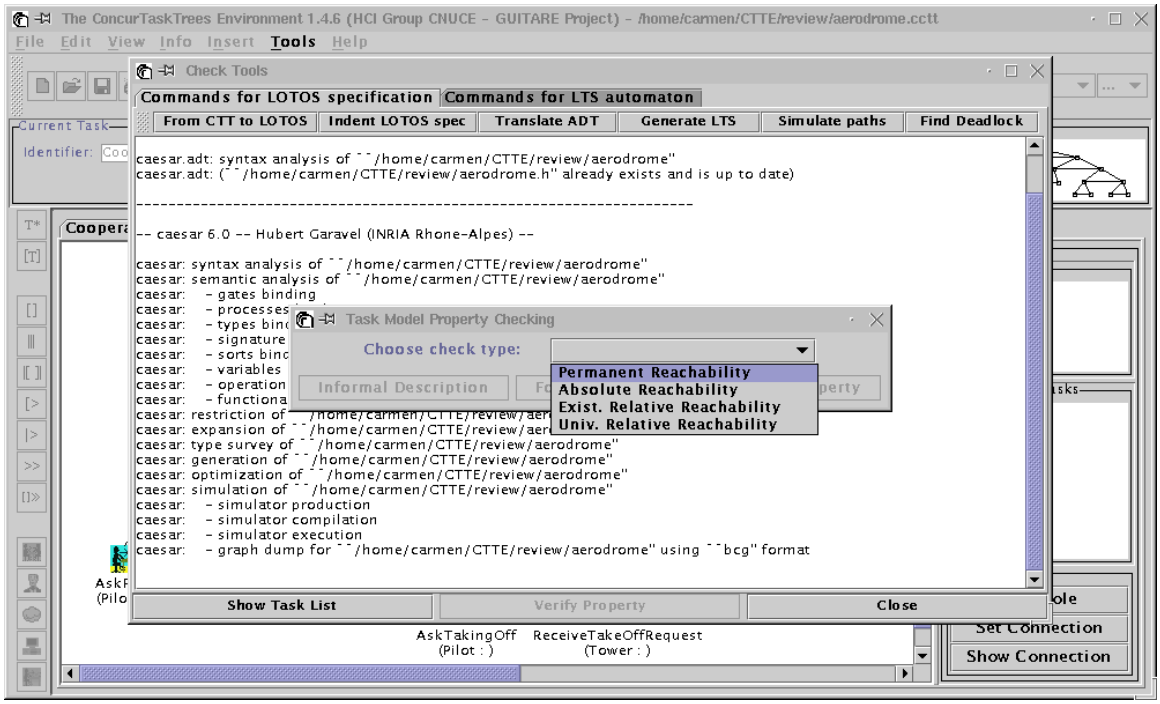

Figure 30. Selection of the Type of Property to Check

The other panel provides commands to handle Labelled Transition Systems (LTS) associated with the current task model (show LTS, edit LTS, and so on).

The first action the user is supposed to do is to activate the transformation from the CTT task model into the corresponding LOTOS specification by selecting the related button (*From CTT to LOTOS*) in the tabbed panel. The CTT environment will show a window where the corresponding LOTOS expression (see Figure 31) is displayed. Such a LOTOS specification can be saved into a separate file and can be used to generate the relative automaton on which the properties will be verified.

To facilitate the specification of the properties, the tool provides templates for a small set of predefined properties that can be filled interactively by selecting the tasks directly within the task model. In addition, even before filling the text fields with actual tasks, the user can obtain information on the various properties. In fact, the tool is able to provide designers with both an informal description and a formal description of each property. The first one (see *Informal Description* button in Figure 32) is intended to make users intuitively understand the meaning of a property without taking care of details of its specification written in the formal language supported by the model-checker underneath. The second one (*Formal Description* button, Figure 32) is for users interested to know the formal specification of the property, which is passed to the underlying model-checker to verify if the specified property holds in the system specified.

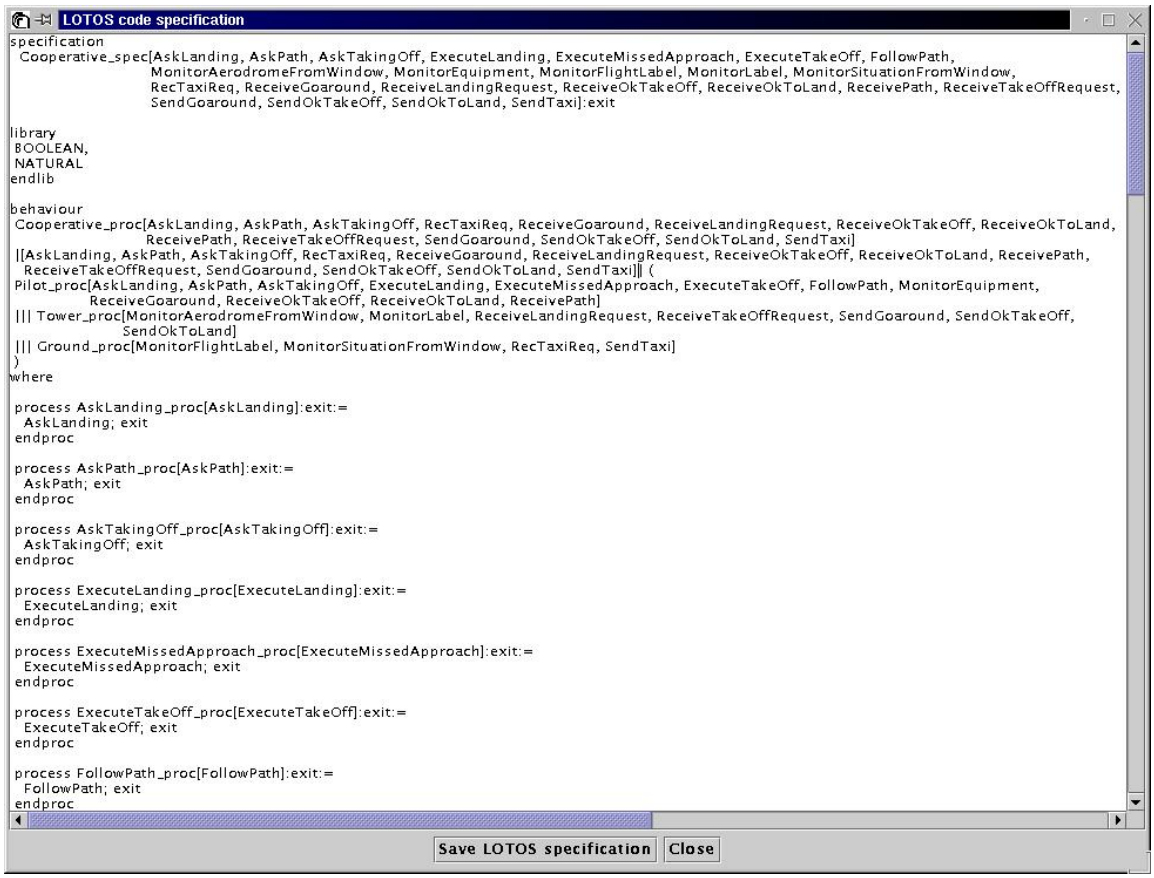

Figure 31. An Example of LOTOS Specification Automatically Derived

Depending on the specific property that the users want to verify, the user interface provides an indication of which information designers should supply. For example, if users want to verify if it is possible to perform a specific task in any state of the system (“Absolute reachability”) they should specify exactly *one* task, and the same holds for “Permanent reachability” property ("Is it always possible to reach a state where a task can be executed?"). On the other hand, if they want to verify if, after having executed a task at least *one* possible evolution exists where another task can be executed ("Existential Relative Reachability") they have to specify *two* tasks. In case of "Universal Relative Reachability" they want to verify if the execution of the second task is possible in *all* the possible next temporal evolutions of the first task performance: again, two tasks are required.

The user interface changes according to the selected property. This is obtained through the introduction of property *templates*. Such constructs help designers while verifying a property and at the same time they prevent them from doing syntax errors, which often occurs in this kind of activity. Instead of directly making the task names involved in the property concerned, now users have just to graphically select one (or more) task icons within the editor panel. This action will result in automatically filling in the text fields corresponding to both task name and associated role (see Figure 32): depending on the structure of each property the dialog will require the user to fill in one or two (text) fields.

This approach, apart from improving the easiness and effectiveness of specifying properties strongly decreases the possibility of common slips, mistakes and inconsistencies due to a wrong selection of task and corresponding roles. In addition, it shows great flexibility because it allows users to specify in the same way both single-user and multi-user properties.

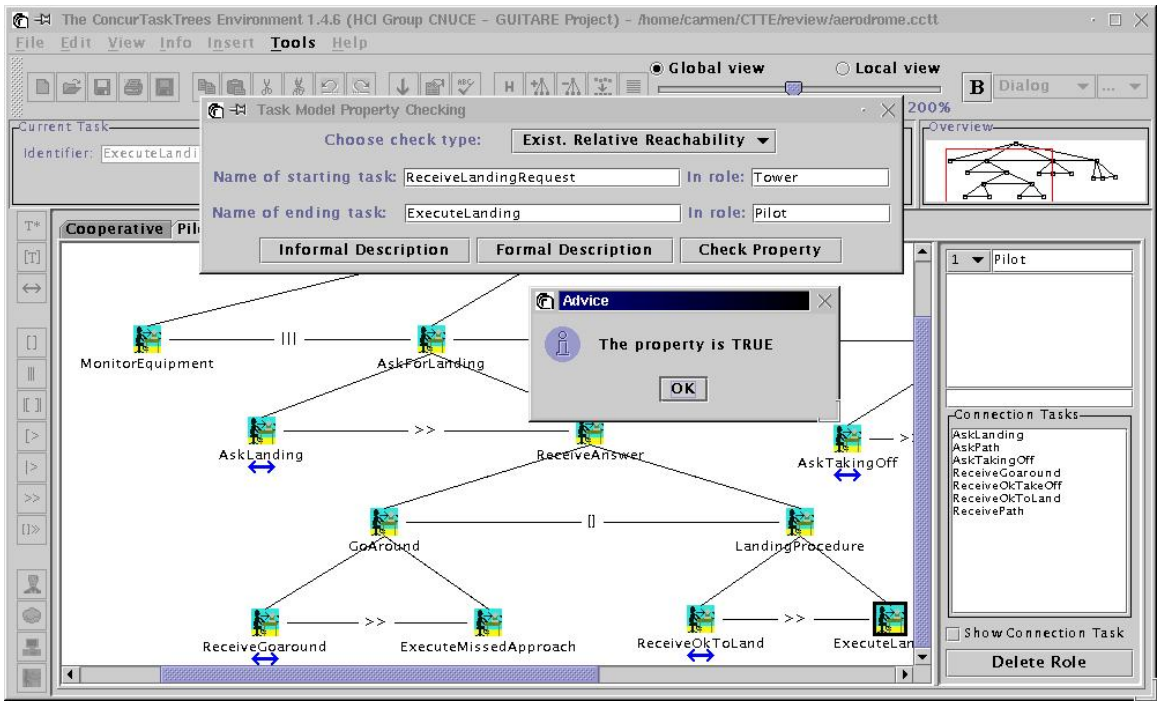

Figure 32. An example of Property Verification in the New Environment

6.7. AN EXAMPLE OF APPLICATION

In this section we analyse a very small example in order to exemplify how the different rules that we just explained in the previous sections could be operatively used. A more extensive example drawn from the a case study in the Air Traffic Control domain area will be described in Chapter 9. Consider the CTT cooperative task model shown in Figure 33. As you can see, the cooperative task model consists of two roles, (Role 1 and Role2), whose activities are detailed in the related single task models. Moreover, one cooperative task model is devoted to describe how such activities synchronise each other (an interleaving operator is used, meaning that the order in which the activities are carried out is irrelevant).

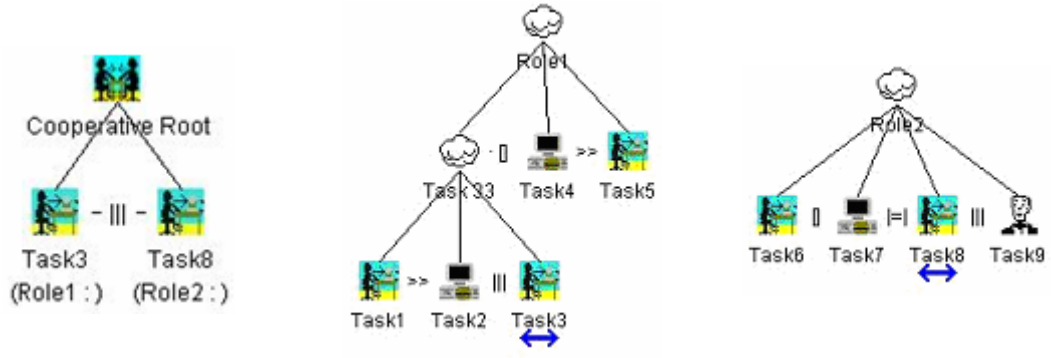

Figure 33. An Example of a CTT Cooperative Task Model

As far as the translation of this cooperative task model is concerned, the process associated with a cooperative task model specification results in concurrently combining (over the cooperative tasks Task3, Task8) the process associated with the cooperative part (Cooperative_Root_proc[Task3, Task8]) and the behaviour expression resulting from the concurrent execution of the processes associated with each single-user task model, which is:

(
Role1_proc[Task1, Task2, Task3, Task4, Task5]
|||

Role2_proc[Task6, Task7, Task8, Task9]
)

The additional constraints expressed in the cooperative part of the task model are specified by the *connection tasks*: they are tasks carried out by one user but taking part also in the execution of a multi-user activity (see tasks Task3, Task8 in Figure 34). For this reason, rather than be connected by a pure interleaving, the process associated to the cooperative part synchronises over the connection tasks with the process resulting from the concurrent execution of the processes associated with each single-user task model (see Figure 34).

```
specification  Cooperative_Root_spec[Task1,  Task2,  Task3,  Task4,  Task5,
Task6, Task7, Task8, Task9]:exit
behaviour
   Cooperative_Root_proc[Task3, Task8]
   |[Task3, Task8]|
   (
    Role1_proc[Task1, Task2, Task3, Task4, Task5]
    |||
    Role2_proc[Task6, Task7, Task8, Task9]
   )
where

process Cooperative_Root_proc[Task3,Task8]:exit:=
   Task3_proc[Task3]
   |||
   Task8_proc[Task8]
endproc

process Role1_proc[Task1, Task2, Task3, Task4, Task5]:exit:=
   Task_33_proc[Task1, Task2, Task3]
   []Task4_proc[Task4]
   >>Task5_proc[Task5]
endproc

[…]

endspec
```

Figure 34. The LOTOS Translation of the Example

Over such a specification it is possible to verify some properties, along the lines of the property templates illustrated in the previous sections. For example, if we wish to verify the *Existential Relative Reachability* over Task1 and Task5 of Role1 in this case we want to check if, after having performed Task1, at least one possible temporal evolution exists in which it is possible to perform (U operator) Task5. By instantiating the template appropriately we obtain the following property:

AG [*Task1*] E[true {true} U {*Task5*} true]

In this case the property is true because, as you can see from the task model specifications, such a path exists and it is represented for instance by this sequence: Task1, Task2, Task3, Task5.

7. Integrating Task Models and System Models

Summary

This chapter presents a set of tools supporting the development of interactive systems using two different notations, CTT and ICO. CTT (and its related environment) has been extensively described in Chapter 3. The other notation, called Interactive Cooperative Objects (ICO), is used for system modelling. Even though these two kinds of models represent two different views of the same world (users interacting with interactive systems), they are built by different people and used independently. The aim of this chapter is to propose the use of scenarios as a bridge between these two views. On the task modelling side, scenarios are seen as possible traces of activities, while on the system side, they are viewed as traces of actions. We propose and discuss an architecture (developed in collaboration with University of Toulouse) for bringing task modelling and system modelling together.

7.1. INTRODUCTION

If the success of a system should be measured by how well it supports the users in accomplishing their tasks, it has been realised at the same time that the usability of a system could be improved if also system modelling is incorporated into early design process and used together task modelling in a combined approach. Actually, task models and system models offer two different views of the same world, with some interesting relationships and intersections between the aspects described by both of them –the actions that can be performed at the user interface level– although each model also captures aspects that are not represented in the other one (for instance, the task model also describes cognitive user activities, whereas the system model contains a description of the system structure). Hence, it is important to analyse the relationships existing between such models, especially considering what possible advantages can be gained from combining such views.

As far as the system modelling is concerned, the object-oriented approach has become the paradigm of choice. As it considers the objects, their properties and how they communicate, object oriented development is largely system centred, rather weak in capturing the users' usage and interaction properties, all of which are critical to the usability of the final system from the user's perspective. Task analysis and modelling, on the other hand, as we have extensively highlighted, provide a user centred view inasmuch as they are exactly geared to describe users' tasks. Then, a combination of them seems highly called for, with several advantages e.g. in checking models' compatibility, for instance ensure that:

- All the objects in the tasks model are part of the data model of the system model
- All the actions in the tasks model are offered by the system model
- All the actions in the system model exist in the tasks models
- All the sequences of actions in the tasks model are "legal" in the system model

However, to integrate the two models effectively, a proper mechanism has to be in place. One of the pre-requisites is to have a software tool for supporting task analysis and modelling, and another critical factor is how to feed the result of task modelling into subsequent system design. Indeed, one of the strengths of a notation is the possibility of supporting it through automatic tools that can help designers to get information from the models. Some research prototype was developed years ago to show the feasibility of the development of such tools, however the first prototypes were rather limited in terms of functionality and usable mainly by the people who develop them. Only in recent years some more engineered tools have been developed, in some cases they are also publicly available. For example, Euterpe (van Welie et al. 98) is a tool supporting GTA (Groupware Task Analysis) where task models are developed in the horizontal dimension with different panels to edit task, objects, actors. A simulator of task

models of single user applications has been given with the support of an object-oriented modelling language (Biere et al. 99). Mobi-D (Puerta & Eisenstein 99) and Trident (Bodart et al. 94) are examples of tools aiming to use information contained in models to support design and development of user interfaces. In particular, in Mobi-D the designer can choose different strategies in using the information contained in task and domain model to derive the user interface design.

In our work we envision a solution based on the use of two tools (CTTE and PetShop) developed to support two different types of models. The former is CTTE, the latter, developed at the University of Toulouse, supports system models described using Petri nets in an object-oriented environment through the ICO notation, which will be described later on. PetShop is able to support editing of a Petri Net controlling the dialogues of a user interface even at run-time thus allowing dynamic change of its behaviour. Their integration allows thorough support to designers since early conceptual design until evaluation of a full prototype.

In this chapter we present an approach aims to overcome this limitation by showing and discussing how we have reached the integration of a set of tools for task modelling with a set of tools for user interface system modelling through the use of abstract scenarios. The goal of such integration is to provide designers with an environment enabling them to see, for example, how a sequence of user tasks can be related to the specification of the behaviour of the underlying system's interconnecting components.

7.2. SYSTEM MODELLING

The main goal of systems modelling is to represent the system data and related actions. More specifically, it represents the behaviour of the system, which should include representing *what* actions are offered by the system to the ser, *when* an action is available (according to the state of the system), what is the *effect* of an action on the state of the system. In other words, it represents both how the system is presented to the user and how the user interacts with it. In this work we consider system modelling done using the Interactive Cooperative Objects (ICO) formalism and related development environment is called PetShop. ICO is based on Petri Nets and O-O approach: OO constructs for structuring, Petri nets for dynamics. Before going into further details, in the next section we provide the reader wit a basic introduction to Petri Nets.

7.2.1. Petri Nets (PN)

Petri nets were introduced by Carl Adam Petri in the early 1960s as a tool for modelling and analyse distributed event-based systems. With Petri Nets, a system is described in terms of an oriented graph with state variables (called *places*) and state-changing operators (called *transitions*), connected by annotated arcs, which can be of two kinds: *input arcs* that connect one place to one transition, *output arcs* that connect one transition to one place. At any time the distribution of tokens among places defines the current state of the modelled system. A transition is said to be eligible if all its input places contain (at least) one token. State changes result from the firing of transitions, yielding a new distribution of tokens. Transition firing involves two steps: (1) tokens are removed from input places and their values bound to variables specified on the input arcs, and (2) new tokens are deposited in the output places with values determined by emission rules attached to output arcs. A transition is enabled to fire when all its input places contain tokens, and the value of such tokens satisfy the (optional) Boolean constraints attached to the input arcs.

Petri Nets allow designers to describe basic constructs in concurrent programming, such as sequence, condition, iteration, parallelism and synchronisation. Due to the limitation of the early model (e.g. no data modelling) more powerful versions have been proposed, moving from 'simple' Petri nets, to Coloured nets, to Petri Nets with Objects (PNO), to timed nets, stochastic nets, etc. In Coloured Petri Nets, the tokens are differentiated through different colours as they assume values from predefined types. Coloured nets allow the modelling of complex systems and data flows. Transition eligibility depends then on the availability of an appropriately

coloured token in all the input places of this transition. Similarly, the output of a transition is not just a token but a specifically coloured token.

In Petri Nets with Objects (PNO), objects are used both for structuring and data modelling. PNO is a Petri Net model where tokens can hold values and more precisely references (in object-oriented environments they can be object identifiers). In particular, ICO (Palanque and Bastide, 1994) is a formalism for describing concurrent object classes with a behaviour and a user interface part. In these classes there are four components: attributes and methods (as in classical object-oriented approaches), a behaviour (described using a PNO) and information in order to take into account the user interface. ICO has the notion of user service which is an operation of the system that can be interactively triggered by the user through some user interface device. In addition, there is also the notion of rendering function which describes how the state of the system is made perceivable to the user.

Petri Nets have been shown to be effective and easy to be understood for small specifications because with them it is easy to follow how the dynamic behaviours can evolve. Problems can arise when the specification is large because it can be difficult to interpret the meaning of a large number of interconnecting arcs and it can be difficult to modify them.

7.2.2. Interactive Cooperative Objects (ICO)

The ICO formalism is basically the continuation of early work on dialogue modelling using high-level Petri nets (Bastide and Palanque, 1990). The various components of the formalism are introduced informally hereafter. A complete and formal presentation of it can be found in http://lihs.univ-tlse1.fr/palanque/Ps/ICOFormalDef.pdf. ICO is a formal notation dedicated to the specification of interactive systems. It uses concepts borrowed from the object-oriented approach (dynamic instantiation, 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.

ICOs are used to provide a formal description of the dynamic behaviour of an interactive application. An ICO specification fully describes the potential interactions that users may have with the application. The specification encompasses both the "input" aspects of the interaction (i.e. how user actions impact on the inner state of the application, and which actions are enabled at any given time) and its "output" aspects (i.e. when and how the application displays information that is relevant to the user). An ICO specification is fully executable, which gives the possibility of prototyping and testing quickly an application before it is fully implemented. The specification can also be validated using analysis and proof tools developed within the Petri nets community.

An ICO model of a system is made up of several communicating objects, where both behaviour of objects and communication protocol between objects are described by Petri nets. 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 a "server" whose role is to execute the service. In the ICO formalism, an object is an entity featuring four components: *services* (or *interface*), *behaviour*, *state* and *presentation*.

Services (or Interface)

The *interface* specifies at a syntactic level the *services* that a client object can request from a server object that implements this interface. The services define the interface (in the programming language meaning) offered by the object to its environment, namely the services supported and their signature: a list of parameters with their type and parameter-passing mode, the type of the return value, the exceptions that may possibly be raised during the processing of the service. For describing this interface the CORBA-IDL language (Object Management Group, 1998) is used. An ICO offers a set of services. In the case of user-driven application, this environment may be either the user or other objects of the application. Indeed, using the ICO

formalism, two kinds of services are identified: services offered to the user, called user services, and services offered to other objects.

Behaviour

The *behaviour* of an ICO defines how the object reacts to external stimuli according to its inner state. This behaviour is described by a high-level Petri net called the Object Control Structure (ObCS) of the object. In ICO formalism, tokens may hold conventional values (integer, string, etc.) or references to other objects in the application. A transition may feature a *precondition* that is a Boolean expression involving the variables labelling the input arcs of the transition. When a transition is fired, one token is removed from each of its input places, and one token is put in each of its output places. A transition features an *action*, which may request services from the tokens involved in the occurrence of the transition, or perform algorithms manipulating the values of tokens.

State

The *state* of an ICO is the distribution and the value of the tokens (called the *marking*) in the places of the Object Control Structure. As services are related to transitions, this allows defining how the current state influences the availability of services, and conversely how the performance of a service influences the state of the object.

Presentation

The *presentation* part of an object states its external appearance, and it is a structured set of *widgets* organized in a set of windows. The user - system interaction will only take place through those widgets. Each user action on a widget may trigger one of the ICO's user services. The relation between user services and widgets is fully stated by the activation function that associates the service to be triggered to each couple (widget, user action). The rendering function is in charge of presenting information according to the state changes that occur. It is thus related to the representation of states in the behavioural description i.e. places in the high-level Petri net.

7.2.3. PetShop Environment

The tool-support for models for user interface has been provided in Pet-Shop, which is an environment allowing development of a technique combining Petri Nets and object-oriented techniques (Bastide and Palanque, 1999). The dialogue of the user interface is modelled by Petri Nets; more precisely, for each widget in the user interface, there is a Petri Net modelling its dynamic behaviour. Such a tool has shown to be useful for modelling the runtime user interface behaviour. It also supports the possibility of connecting the user interface and the related Petri Net and shows how the tokens evolve while the user interacts with the application. Thus, the purpose is to support the analysis of the user interface system, rather than the related task model: it provides a description of the objects making up the user interface and the underlying system and how they behave, whereas, in the task model, there is a description of the logical activities supported.

In this section we introduce the PetShop environment and the design process it supports. The interested reader can find more information in (Sy *et al.*, 1999). Figure 35 presents the general architecture of PetShop. The rectangles represent the functional modules of PetShop. The documents-like shapes represent the models produced and used by the modules.

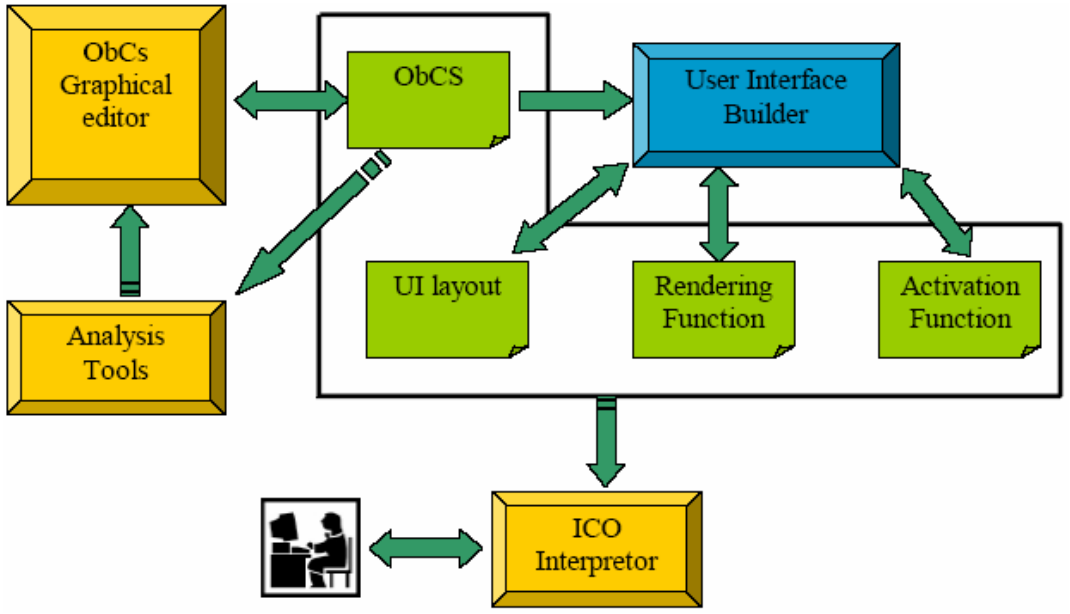

Figure 35. Architecture of PetShop Environment

Thanks to the use of high-level Petri nets for the description of the behaviour of the classes, an ICO formal description is executable through a Petri net interpreter. Besides, high level ICO description can be refined by adding details like associating some extra code to the ObCS's transitions and coding the rendering methods in the widget part that are called by the rendering function. When refined enough, an ICO specification becomes prototypable as at that stage the Petri net interpreter can also monitor application code while executing the specificaiton. Indeed, when running the ObCS it produces a sequence of events that describes the state changes of the application and when this sequence is handled it produces changes on the presentation part through the activation and rendering function. PetShop features an Object Petri net editor that allows for editing (see Figure 36) and executing the ObCSs of the classes.

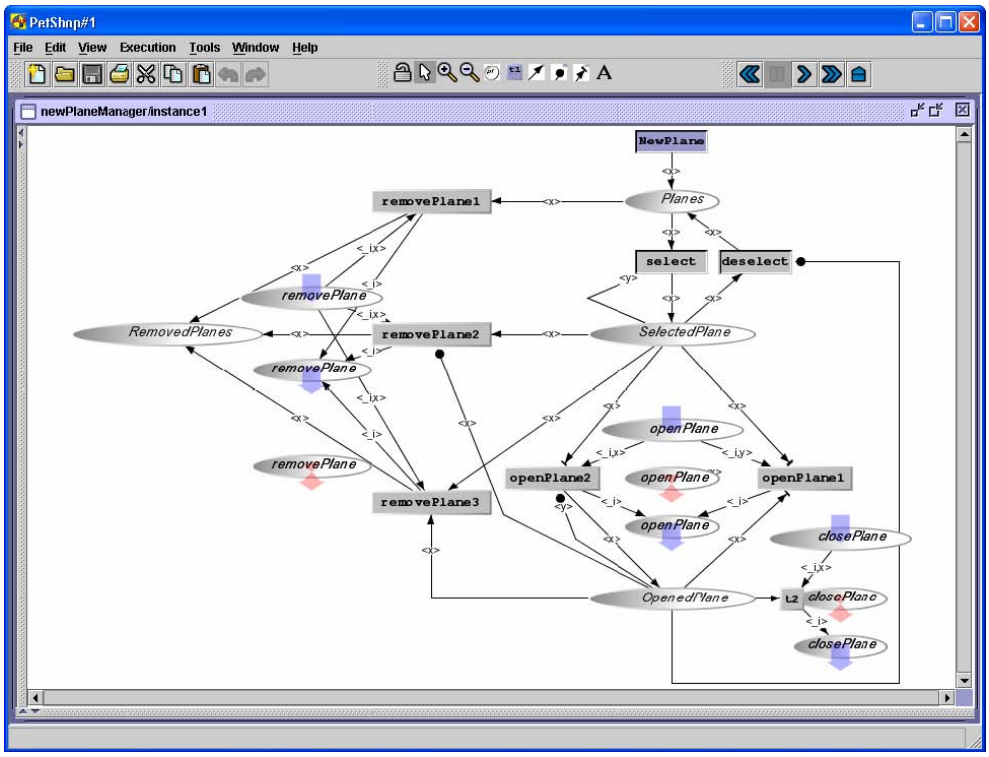

Figure 36. The ObcS Editor in PetShop Environment

All the components of the ObCS can be interactively built using PetShop. The editing of the Object Petri net is done graphically using the Palette in the centre of the toolbar. The left part of the toolbar is used for generic functions such as load, save, cut-copy and paste. The right hand side of the toolbar drives the execution of the specification.

A well-known advantage of Petri nets is their executability. This is highly beneficial to our approach, since as soon as a behavioural specification is provided in term of ObCS, this specification can be executed to provide additional insights on the possible evolutions of the system. Figure 37 shows the execution of the specification of the Plane Manager in Petshop. The ICO specification is embedded at run time according to the interpreted execution of the ICO.

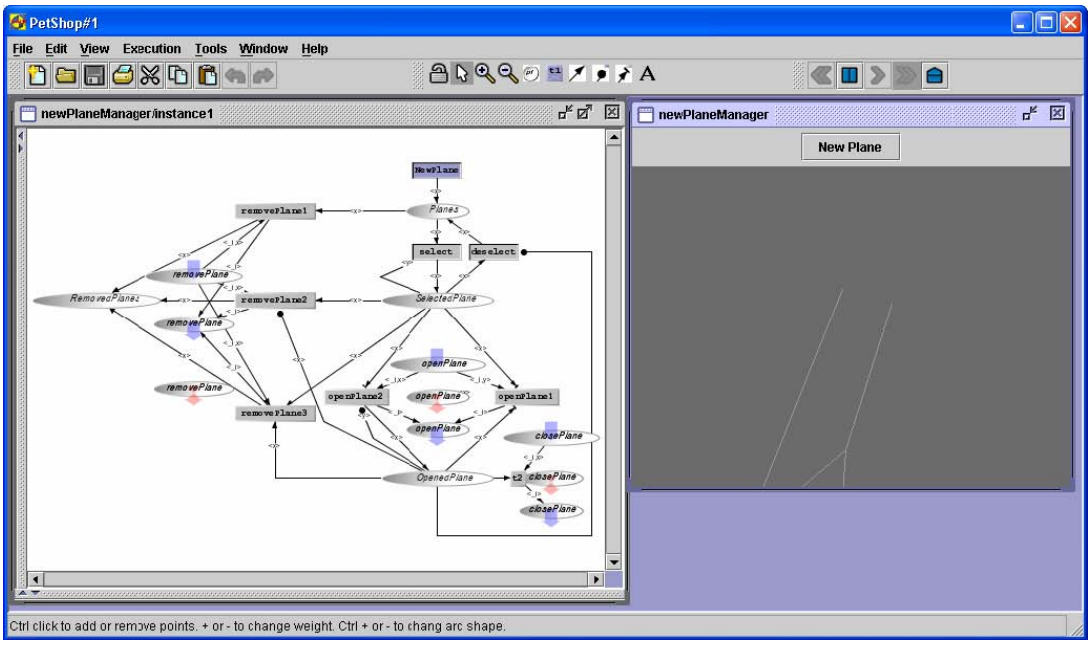

Figure 37. Execution of ICO Specification

At run time, the user can both look at the specification and the actual application. They are in two different windows overlapping in Figure 37. The window PlaneManager corresponds to the execution of the window with the Object Petri net underneath. In this window we can see the set of transition that are currently enabled (represented in dark grey and the other ones in light grey). This is automatically calculated from the current marking of the Object Petri net.

Each time the user acts on the PlaneManager the event is passed onto the interpreter. If the corresponding transition is enabled then the interpreter fires it, performs its action (if any), changes the marking of the input and output places and perform the rendering associated (if any).

7.3. THE INTEGRATION FRAMEWORK

Task and system models are particularly important when designing and developing interactive software systems. In both industrial and academic communities there is a wide agreement on the relevance of task models as they allow expressing the intentions of the users and the activities they should perform to reach their goals. These models also allow designers to develop an integrated description of both functional and interactive aspects. Within the development lifecycle of an application the task-modelling phase is supposed to be performed after having gathered information on the application domain and an informal task analysis phase. The result of the latter one is an informal list of tasks that have been identified as relevant for the application domain. After this step, in developing a task model designers should be able to clarify many aspects related to tasks and their relationships. In some cases task models are first developed and then used to drive the system design. In other cases designers have to address an existing system and need to develop a task model to better understand and evaluate its behaviour (Palanque *et al.*, 1997).

System models describe important aspects of the user interface. In this work we pay particular attention to the dialogue supported by the system: how user actions and system feedback can be sequenced. Scenarios (Carroll 1995) are a well-known technique in the human-computer interaction area. They provide a description of one specific use of a given system, in a given context. They are an example of usage. Their limited scope is their strength because they can easily highlight some specific aspect and are easily understood and remembered. Thus, they can also be considered as a useful tool to compare different models and analyse their relationships.

One point is that we can check if the task models fulfil the expected requirements and if the system model matches the planned behaviour. However, what cannot be missed is checking if both two models are consistent, which means if both specifications really refer to the same system. This requires checking if for every user action assumed in the system model there is an actual counterpart in the task model, and each system output provided to the user has been foreseen in the task model specification.

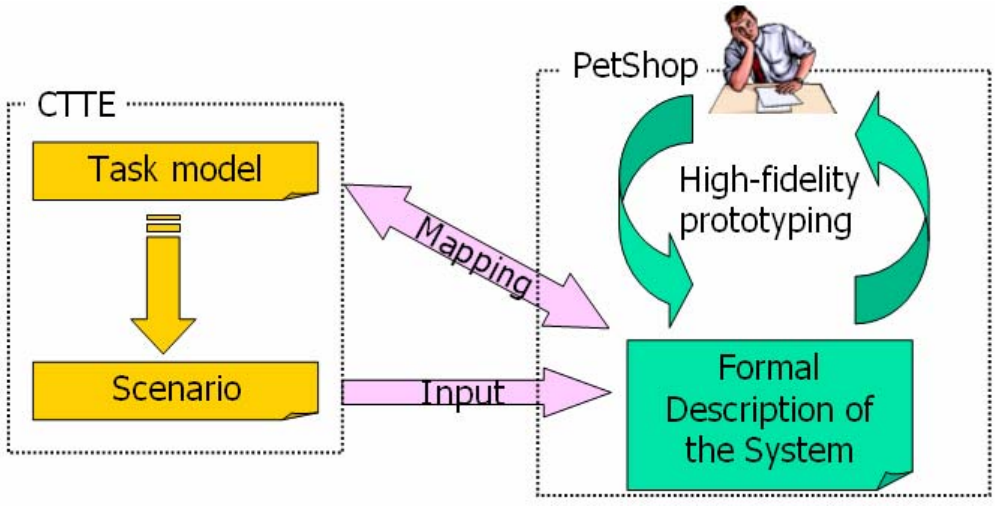

Figure 38. The Flow of Information between CTTE and PetShop

Another relevant point that has to be highlighted is that these two models can be specified by different people and in distinct moments of the design cycle of the user interface development process. Indeed, especially in real case studies sometimes the task models will be developed at first, sometimes they might be specified after the system model has already been obtained. So, we need an approach that does not have specific constraints and requirements on what is assumed to be available at a certain phase of the system design, as it can be equally used efficiently in both cases. In our approach, the task model will be used to automatically generate use cases and scenarios, which in turn Such will be used as the common "lingua franca" to ensure that there is actual correspondence between what has been specified within the task model and what has been specified in the description provided by the system model (see Figure 38). The idea is to focus the attention on specific examples of usage either on the system side or on the tasks side and to check if on these simple examples of the system use such correspondence exists.

Considering the task model-side, in our approach we used the ConcurTaskTrees notation for specifying tasks. The semantics of the operators used in this notation has been described in Section 3.3. This notation allows users to explicitly express how the allocation of the different tasks has been assumed in the system design. Such allocation could be on the user alone (user tasks), on the application alone (application tasks), on the interaction between the user and the application (interaction tasks), or if the activity is too general to be specifically allocated on each of them we use "abstract" task. This aspect proves to be effective especially when both comparison and integration of different models has to be carried out, such as in our case. The notation provides the ability to specify in the task model when system support is requested on the user interface. This allows comparing and cross-checking if the task model reflects and is adequately supported by the corresponding system model.

More specifically, the points that have to be carefully checked in the task model specification are the interaction and application tasks. Application tasks indicate that at a certain point during a session a specific behaviour of the system is expected. This behaviour can be expressed in terms of a specific feedback of an action the user has performed while interacting with the system; in terms of a result the system has produced after some elaboration; in terms of availability of a specific input needed to users in order to perform their tasks. All those

possibilities have to be carefully supported especially if the considered domain is vast and complex as the air traffic control field considered in our case study. Such domain is composed of a number of entities that maintain a complex relationship structure, due to their internal structure and to the dynamic behaviour they follow, which has to be appropriately presented to the users.

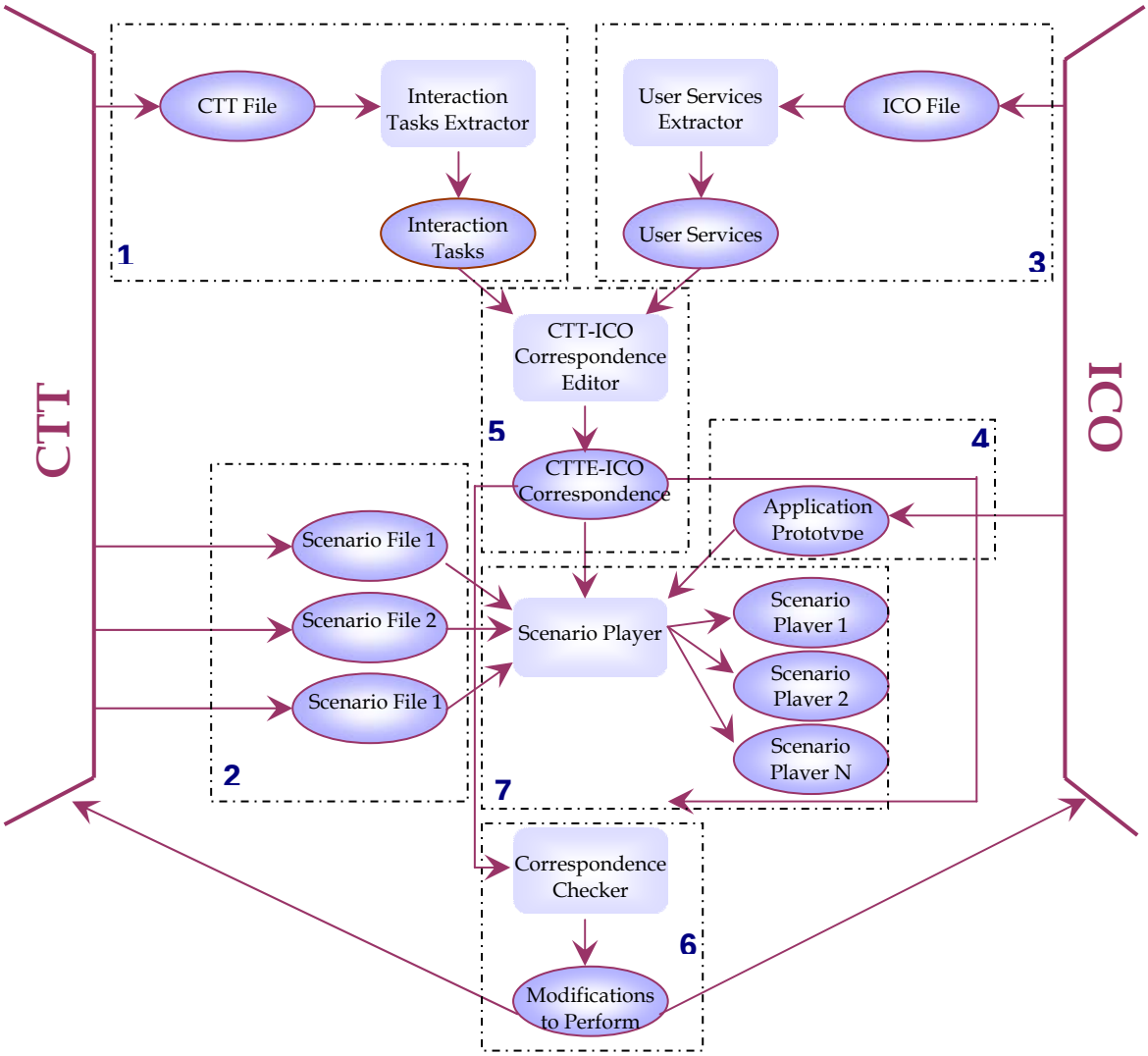

Figure 39. The Framework for CTTE– PetShop (or CTT-ICO) Integration

The integration framework takes full advantage of the specific tools that were initially developed in a separate manner. One advantage of this separation is that it allows for independent modification of the tools, provided that the interchange format remains the same.

The two notations model slightly different aspects: CTT is a notation for task modelling whereas ICO is a notation for specifying concurrent systems, thus an automatic conversion from one notation to the other one would have been difficult. We have preferred a different solution that is easier to implement and better refers to the practise of user interface designers. Indeed, often designers use scenarios for many purposes and to move among the various phases of the design cycle. So, they can be considered a key element in comparing design solutions from different viewpoints.

The different parts of the framework for CTT-PetShop integration are shown in Figure 39 and referred by means of numbers.

CTT Environment

In Chapter 3 we have extensively described the main features of the CTT Environment in supporting modelling, analysis and simulation of task models. For the purpose of integration we only use the interactive tool for editing the tasks and the simulator, which allows designers to interactively simulate the behaviour specified in the task model and build scenarios accordingly. Thus the two main outputs provided by CTT environment and their processing are a set of task models and a set of scenarios. These two sets are exploited in the following way: from the ConcurTaskTrees specification a set of interaction tasks is extracted which represents a set of manipulations that can be performed by the user on the system (part 1 of Figure 39), the set of scenario is used as is by the integration tool (part 2 of Figure 39). They are highlighted on Figure 39 as part 1 and part 2.

ICO Environment

On the ICO's side, amongst the features of the related environment (PetShop) presented in section 7.2.3, the one that is used for integration purposes is the tool for editing and executing system models. In fact, from this specification we extract a set of user services and from the ICO environment we use the prototype of the system modelled. In Figure 39 such two main outputs of the ICO environment and their processing are highlighted as part 3 (user services) and part 4 (prototype).

As it was already said, a user service is a set of particular transitions that represents the functionalities offered to the user by the system. These transitions are performed when and only when they are fireable and the corresponding user actions are performed (which is represented by the activation function in the ICO formalism).

The Correspondence Editor

The activities that are managed by the correspondence editor correspond to part 5 and part 6 of Figure 39. The first component of the correspondence editor relates interaction tasks in the task model to user services in the system model (part 5 of Figure 39). When the task model is refined enough, the leaves of the task tree represent low-level interactions on the artefacts. It is then possible to relate those low-level interactive tasks to user actions in the system model that are represented, in the ICO formalism, by user services.

In order to check that this correspondence is valid a model checker (part 6 of Figure 39) was developed. The properties checked by the model checker correspond to the verification and validation phase in the development process. Validation phase relates to the question "do we have modelled the right system?" while the verification phase address the question "do we have modelled the system right?". In the context of ICO-CTT integration for the verification phase the model checker addresses the following two questions:

- Are there at least as many user services in the ICO specification as interaction tasks in the CTT model ?
- Are all the possible scenarios from the task model available in the system modelled ?".

In the context of ICO-CTT integration for the validation phase the tool addresses the following two questions:

- Are there more user services in the ICO specification than interaction tasks in the CTT model ?"
- Are there scenarios available in the system model that are not available in the task model?".

If the answer is yes for one of these two sub rules, the system modelled offers more functionalities than expected by the task model described with CTT. This leads to two possible

mistakes in the design process. Either the system implements more functions that needed or the set of task models built is incomplete. In the former case the useless functionalities must be removed. In the latter case either task models using this functionality are added or the use of this functionality will never appear in any of the scenarios to be built.

The role of the correspondence checker is to notify any inconsistency between the CTT and the ICO specifications. In this part a CTT-ICO correspondence file that stores the mapping between elements in the task and system models is produced.

Execution: the Scenario Player

As a scenario is a sequence of tasks and as we are able to put a task and a user service into correspondence, it is now possible to convert the scenarios into a sequence of firing of transitions in the ICO specification.

An ICO specification can be executed in the ICO environment and behaves according to the high-level Petri net describing its behaviour. As the CTT scenarios can be converted into a sequence of firing of transitions, it can directly be used to drive the execution of the ICO specification. To this end a tool dedicated to the execution of an ICO formal description of a case study driven by a scenario extracted from a task model (see Part 7 of Figure 39) was developed.

8. Task model –based Evaluation of Interactive Systems

Summary

In this chapter we discuss how to perform a systematic inspection-based analysis to improve both usability and safety aspects of an application. The analysis considers a system prototype and the related task model and aims to evaluate what could happen when interactions and behaviours occur differently from what the system design assumes. We also provide a description and discussion of an application of this method to a case study in the air traffic control domain. The combination of basic and high-level tasks with the many types of deviations considered allows designers to address a broad set of issues. This can be useful to analyse how the system design supports unexpected deviations in task performance. Such analysis is particularly important in interactive safety-critical systems where user error may even threaten human life.

8.1. INTRODUCTION

In recent years, a number of usability evaluation techniques have been developed. They can be classified according to many dimensions –for instance, the level of refinement of the user interface considered or the amount of user involvement in the evaluation. More generally, usability evaluation methods can be divided into different categories: *model-based approaches*, where a significant model related to the interactive application is used to drive the evaluation; *inspection-based assessment*, where some expert evaluates the system, or some representation of it, according to a set of criteria; and *empirical testing* where direct use of the system is considered.

In this chapter we discuss how task models can be used in an inspection-based usability evaluation for interactive safety-critical applications. To this end, we first introduce our approach to describe how to use information contained in task models to support an exhaustive inspection-based evaluation. Then, in order to exemplify the approach, we discuss a sample example where the method is applied. A more extended example will be described in Chapter 9.

8.2. TASK MODELS AND USABILITY EVALUATION

The use of task analysis and modelling has long been applied in the design of interactive applications. However, less attention has been paid to their use to support systematic usability evaluation. Task models can also be useful in supporting design and evaluation of interactive safety-critical applications.

Task models can be useful in various phases of the design cycle. They can play an important role in the requirements elicitation and specification phase (see for example GTA in (van der Veer et al, 1996)), for example, by requiring precise definition of temporal relationships between the different activities that should be performed, avoiding any ambiguities. In addition, task models can be used to support the design of interactive applications: a number of criteria have been identified on how to use the information contained in CTT task models to drive the design of the user interface. Task models satisfy the need to maintain concise and precise documentation of the system under consideration.

They can also be useful in the evaluation phase. The work done so far in model-based evaluation has mainly aimed to support performance evaluation (such as GOMS approaches) to predict task completion times or usability evaluation through the automatic analysis of user interactions (Ivory et al, 1999). An example of the latter techniques can be found in (Lecerof and Paternò, 1998), where the use of task models in remote usability evaluation is described: the possible activities supported by the application and described by the task model are used to analyse real user behaviour as revealed by automatically recorded log files of user interactions with a graphical interface. This type of evaluation is useful for evaluating final versions of applications. However, we think that task models may also help support evaluation in the early

phases of the design cycle. Indeed, inspection-based methods are often applied to evaluate prototypes in order to give suggestions for obtaining improved versions. They are less expensive than empirical testing and one advantage is that their application at least decreases the number of usability problems that can be detected by the final empirical testing.

A number of inspection-based evaluation methods have been proposed. In heuristic evaluation (Nielsen, 1993) a set of general evaluation criteria (such as visibility of the state of the system, consistency, avoid providing useless information, ...) are considered and evaluators have to check whether they have been correctly applied. This method heavily depends on the ability of the evaluator and many software engineers may have problems to understand how to apply such general criteria to their specific cases. In cognitive walkthrough (Wharton et al, 1994) the evaluators have to identify a sequence of tasks to perform and for each of them four questions are asked. While this method is clear and can be applied with limited effort, in our opinion it has a limitation: it tends to concentrate the attention of the evaluator on whether or not the user will be able to perform the right interactions. Little attention is paid on what happens if users perform errors (interactions that are not useful for performing the current task) and on the consequences of such errors. It is easy to understand how crucial this aspect is in interactive safety-critical applications where the consequences of human errors may even threaten human life.

Few works have addressed this type of problem. Reason (Reason, 1990) introduced a first systematic analysis concerning human error, including the simple slips/mistakes distinction. Human Reliability Analysis (Hollnagel, 1993) stems from the need to quantify the effects of human error on the risks associated with a system and typically involves estimating the probability of the occurrence of errors, which quickly runs into the problem of acquiring the data necessary for reliable quantification. Practical experience shows that different methods in this area often yield different numerical results, and even the same method may give different results when used by different analysts. Leveson (Leveson, 1995) introduced a set of guidelines for the design of safety-critical applications, but little attention was paid to the user interface component. Some guidelines for safe user interactions design are proposed, but they are too general to be used systematically when designing the user interface. Then, research moved on to finding systematic methods for supporting design and evaluation in this area. THEA (Fields et al, 1997) uses a scenario-based approach to analyse possible issues. While our approach involves end-users in the evaluation exercise, THEA supports a judgment study performed by experts. Formal methods (Palanque *et al*., 1997) have also been considered for this purpose and have showed to be useful in analysing limited parts of these applications. In (Galliers *et al*., 1999) the authors propose an analysis that supports re-designing a user interface to avoid the occurrence of errors or to at least reduce their effects. The analysis is supported by a probabilistic model that uses Bayesian Belief Nets. Johnson has developed a number of techniques -see for example (Johnson and Botting, 1999)- for analysing accident reports, which can be useful to better understand the human errors that have caused real accidents.

The contribution of our method resides in the help that it provides to designers in order to systematically analyse what happens if there are deviations in task performance with respect to what was originally planned. It indicates a set of predefined classes of deviations that are identified by *guidewords* (MOD, 1996). A guideword is a word or phrase that expresses and defines a specific type of *deviation*. These types of deviations have been found useful for stimulating discussion as part of an inspection process about possible *causes* and *consequences* of deviations during user interactions. Mechanisms that aid the *detection* or *indication* of any hazards are also examined and the results are recorded. In our approach we build upon the HAZOP family of techniques, originally developed for investigating hazard and operability issues in the chemical process industry. While the basic HAZOP techniques have been used for analysing software-based systems (McDermid et al, 1994) and safety-critical systems (Burns et al, 1993), we note a lack of proposals to introduce them in structured methods for user interface analysis, design and evaluation with the support of the task models.

We use an inspection technique to go systematically through the actions that are required from participants, and consider ways in which failures might arise during a scenario, what the effect of failures might be, and what safeguards and defences exist in the system. The particular questions we will seek answers to are: what are the potential hazards that can arise as consequences of deviations, failures in communication, or erroneous actions in the scenario? What recommendations concerning the user interface design can be provided to mitigate possible hazardous states and their effects?

The basic idea is that for each design option we consider the main tasks and the possible deviations that can occur in the performance of the task. Interpreting the guidewords in relation to a task allows the analyst to systematically generate ways the task could potentially go wrong, as a starting point for further discussion and investigation. Such analysis should generate suggestions as to how to guard against deviations as well as recommendations about user interface designs that might either reduce the likelihood of the deviation or support detection and recovery.

8.3. THE METHOD

In the analysis, we consider the system tasks model: how the design of the system to evaluate assumes that tasks should be performed. The goal is to identify the possible deviations from this plan. Interpreting the guidewords in relation to a task allows the analyst to systematically generate ways the task could potentially deviate from the expected behaviour during its performance. This serves as a starting point for further discussion and investigation. Such analysis should generate suggestions about how to guard against deviations as well as recommendations about user interface designs that might either reduce the likelihood of the deviation or support its detection and recovery from hazardous states.

The method is composed of three steps:

- *Development of the task model of the application considered*; this means that the design of the system is analysed in order to identify how it requires that tasks are performed. The purpose is to provide a description logically structured of tasks to be performed. We use the CTT notation but other notations for task modelling with similar operators could still be suitable.
- *Analysis of deviations related to the basic tasks*; the basic tasks are the leaves in the hierarchical task model, tasks that the designer deems should be considered as elementary units.
- *Analysis of deviations in high-level tasks*, these tasks allow designer to identify *group of tasks* and consequently to analyse deviations that involve more than one basic task. Such deviations concern whether the appropriate tasks are performed and if such tasks are accomplished following a correct ordering.

It is important that the analysis of deviations be carried out by interdisciplinary groups where such deviations are considered from different viewpoints and backgrounds in order to carry out a complete analysis. The analysis follows a bottom-up approach (first basic tasks, and then high-level tasks are considered) that allows designers first to focus on concrete aspects and then to widen the analysis to consider more logical steps. We have found useful to investigate the deviations associated with the following guidewords:

- *None*, the unit of analysis, either a task or a group of tasks, has not been performed or it has been performed but without producing any result. None is decomposed into three types of deviations depending on why the task performance has not produced any result. It can be due to a *lack of initial information*

necessary to perform a task or because *the task has not been performed* or because it has been performed but, for some reason, *its results are lost*. For example, if we consider the task of selecting an aircraft on a graphical screen, it can be not performed because the system does not provide correct presentation of the aircraft, or because the information is correctly presented but the controller does not realise the need for the selection, or because the controller mentally selects the aircraft but then forgets to perform the action on the screen because interrupted by other activities.

- *Other than*, the tasks considered have been performed differently from the designer's intentions specified in the task model; in this case we can distinguish three sub-cases: less, more or different. Each of these sub-cases may be applied to the analysis of the input, performance or result of a task, thus identifying nine types of deviations (*less input, less performance, less output, other than input, other than performance, other than output, more input, more performance, more output*). An example of the less input sub-case is when the user sends a request without providing all the information necessary to perform it correctly. Thus, in this case the task produces some results but they are likely wrong results.
- *Ill-timed*, the tasks considered have been performed at the wrong time. Here we can distinguish at least when the performance occurs *early or late* with respect to the planned activity.

The choice of these guidewords is based on the observation that a task is an activity that usually requires some initial information, then their performance occurs and lastly such performance generate some result. A deviation can occur in any of the parts of a task. n addition, for each task analysed it is possible to store in one table the result of the analysis in terms of the following information:

- *Task*, indicating the task currently analysed*;*
- *Guideword*, indicating the type of deviation considered;
- *Explanation*, explaining how the deviation has been interpreted for that task or group of tasks;
- *Causes*, indicating the potential causes for the deviation considered and which cognitive faults might have generated the deviation;
- *Consequences*, indicating the possible effects of the occurrence of the deviation in the system;
- *Protection*, describing the protections that have been implemented in the considered design in order to guard against either the occurrence or the effects of the deviation on the system;
- *Recommendation*, providing suggestions for an improved design able to better cope with the considered deviation.

Tables can be used as documentation of a system and its design rationale, giving exhaustive explanation of cases when an unexpected use of the system has been considered and which type of support has been provided in the system to handle such abnormal situations. e

think useful to classify the explanation in terms of which phase of the interaction cycle (according to Norman's model) can generate the problem:

- *Intention*, the user intended to perform the wrong task for the current goal. An example of intention problem is while controllers aim to solve an air traffic conflict to obtain a safer state they decide to increase the level of an aircraft but in this manner they create a new conflict.
- *Action*, the task the user intended to perform was correct but the actions identified to support it were wrong. An example of action error is when controller wants to edit graphically a path to send to an aircraft by selecting points that are not selectable.
- *Execution*, the performance of an action was wrong, for example the controller selects a wrong aircraft because of a slip;
- *Perception*, the user has difficulties in perceiving or correctly perceiving the information that is provided by the application. A perception problem is when the user takes long time to find the user interface element necessary to perform the next task.
- *Interpretation*, the user misinterpreted the information provided by the application. An interpretation problem is when there is a "More Info" button but the user misunderstands for what topic more information is available.
- *Evaluation*, in this case the controller has perceived and interpreted correctly the information but it is wrongly evaluate, for example the controller detects an air traffic conflict that does not exist.

Many types of strategies can be followed when a deviation occurs: these range from techniques aiming to prevent abnormal situations, to others that allow such situations could arise, but inform the user of their occurrence in order to either mitigate or aid in recovering from potential problems (if any).

8.3.1. Organisation of the Evaluation

The evaluation exercises should be carefully organised. It is important to involve, besides the evaluators, at least some software developers and real end-users as well. For each stakeholder it would be better to have more than one representative to get a good level of reliance of the results achieved. The participants should be introduced to the method so that they can understand the structure of the exercise and the reasons for it. The meaning of each requested information should be precisely explained, otherwise the entire process might be distorted. For example, the participants could associate a too restrictive meaning to each column, neglecting to mention cases that, on the contrary, are interesting too. A classic example is the interpretation of the "*protection*" word. Protection sometimes is thought of only in terms of the system-side (*automatic* or semi-automatic safeguards provided by the system). On the contrary, we refer to a wider denotation: not only all the automatic protections offered by the system, but also all the protections provided by the other users involved in the system. For example, all the situations when human agents have sufficient information to be able to realise that something is going wrong in the system, this *does* represent a protection. If this is not well understood, such information could be missing.

The evaluators should have the task model available in order to have a representation helping them to systematically identify what should be analysed. In order to simplify the exercise it is not necessary that end users follow the details of the notation used for specifying the task model. It is sufficient that the model is used by the evaluators who analyse it to decide

the questions or issues to raise. In addition, in real case studies it may happen that the task modelling phase is carried out in a time different from when the deviation analysis exercise is carried out. Thus, it can happen that the participant do not completely agree on how some parts of the task models have been modelled because e.g. aspects that are relevant have been neglected or other objections are raised, so it is important to reach a common agreement before starting the exercise.

During the exercise the prototype (at whatever level of refinement it is), should be available and the participants should spend some time to understand its main features. It can be useful to have an audio record of the session that can be considered later on to check some parts of the discussion. Evaluators should drive the discussion raising questions following the task model and the list of deviations. Often in the discussion it is useful to explain which presentation elements of the user interface support the logical task considered. In addition, for each task it is important to clearly identify what information is required to perform it and what the result of its performance is. If users make a mistake in understanding the information requested for each task, their answers might be inappropriate. The worst thing is that those misunderstandings might not be so evident at the beginning: if discovered ahead in the discussion they could invalidate parts of the previously achieved results, inevitably wasting time. The result of sessions can be saved in tables like Table 7.

Tables can take some effort to fill in and in some cases designers may consider furnishing only a meaningful subset of cases able to address all the main issues. The effort required to be exhaustive is justified especially when systematic documentation of the system and design rationale are required. This is particularly important in projects and teamworks in which different people have to consider the system design at different times, for example, when discussing how to satisfy new requirements.

8.4. AN APPLICATION EXAMPLE CASE STUDY

During the Mefisto Project, we had the opportunity to apply the method to a prototype for the air traffic control in an aerodrome, whose main purpose was to support data-link communications to handle aircraft movement between the runways and the gates. Here we have applied the method to an excerpt. More details on the case study will be provided in Chapter 9. To better explain how the method works, we consider an example of both basic task and high-level task.

8.4.1. Evaluating a Low-Level Task

Table 7 presents an example of the analysis of a basic task in an air traffic control case study. The task considered is *Check deviation* (the controller checks whether an aircraft is following the assigned route in the airport). The class of deviation considered is *None*. First, we have to identify the information required to perform the task. In this case it is the state of the traffic in the airport and the path associated with the aircraft under consideration. If we consider the *No input* case, we can note that it has different causes, both generating perception problems: either a system fault has occurred or the controller is distracted by other activities. This shows that the method is able to consider both system and user errors. In any case, the consequence is that the controller has not an updated view of the air traffic.

Task: Check Deviation			**Guideword: None**	
Explanation	**Causes**	**Consequences**	**Protections**	**Recommendations**
No input The controller has not information concerning the current traffic	The system is broken. *Perception* problem	The controller has not an updated view of the air traffic	The controller looks at the window to check the air traffic	Duplication of communication system
	Controller is distracted or interrupted by other activities *Perception* problem		Pilot calls controller to check the path	Provide an automatic warning message only when an aircraft is deviating from the assigned path.
No performance The controller has the information but he doesn't check it carefully	Controller is distracted or over confident *Interpretation* problem		Red line in case of runway incursion	
No output The controller finds a deviation but immediately forgets it	Controller is interrupted by other activities *Intention* problem			

Table 7. Example of Analysis of Task Deviations

The protection in the current system changes accordingly, if the system is broken then the controller has to look through the window in order to check the state of the traffic. If the controller is distracted then pilots, especially if they perceive that something abnormal is occurring, may contact the controller. The recommendations for an improved system change depending on the case considered: duplication policy should be followed against system faults whereas warning messages should be provided in case of aircraft deviating from the assigned path. Slightly different possibilities are considered in the other sub-cases. For instance, in the *No performance* case we consider when the information is available but for some reason the task is not performed: the controller is distracted or over confident, so he does not interpret correctly the information. In the *No output* case we have that the task is performed but its results are lost, e.g. the controllers find a deviation but they forget it because they are unexpectedly interrupted by another activity.

8.4.2. Evaluating a High-Level Task

When considering not basic tasks, the deviations should be applied in a slightly different manner although we can still use similar tables to store the results of the analysis. In fact, in these cases, the interpretation and the application of each guideword to a higher-level task has to be properly customised depending on the temporal relationships existing between its subtasks. For example, if a higher-level task has all its basic tasks connected by the sequential enabling operator, because of the temporal relationship that relates the first left subtask to the others, a lack of input for the first basic task means a lack of input for the whole parent task. Therefore, this case (*None/No input*) of the analysis of the parent task can be brought back to the correspondent case of its first left child. For the other guidewords, the reasoning changes accordingly. For example, the *Less* guideword might mean that one associated sub-task has not been carried out (for example, the user forgets to perform the last subtask), the *More* case might arise because of an additional, unforeseen performance of one subtask (e.g., one task is executed more than once), the *Different* case occurs when a different temporal relationship is introduced within the subtasks (e.g., the subtasks are performed concurrently rather than sequentially) or wrong subtasks are performed. The analysis of *Early* and *Late* guidewords should consider the

parent task as a whole, inquiring about the impact on the system of all the situations when a too early/late performance of the parent task occurs.

Of course, with different arrangement of temporal operators the reasoning has to be appropriately customised. Anyway, what has to be emphasised is that, depending on the specific decomposition of a higher-level task into lower-level tasks, the application of the guidewords has to be adapted on the basis of the particular temporal relationships specified. A systematic analysis can require the consideration of a high number of cases, sometimes similar among them, thus only a selection of meaningful cases can be deeply analysed in order to limit the amount of time and effort spent.

9. A Case Study

Summary

In this chapter the air traffic control domain will be considered and analysed to the aim of providing the basis on which detailed examples of the techniques explained in previous chapters will be shown. Then, this chapter will present a selection of air traffic control case studies that raise interesting research issues form both safety and usability viewpoints, and that will be used to apply the methods and techniques described in the previous chapters in order to illustrate their main novel ideas.

9.1. INTRODUCTION

In the previous chapters we defined the main lines of our approach, also providing some small examples with the purpose of just exemplifying the underlying concepts. In this chapter we move on applying the proposed method to more sizeable settings in the ATC domain. Within the sphere of safety critical applications, the air traffic control domain lent itself to useful analysis inasmuch as the different phases of a flight require different applications to be controlled, with each application raises diverse requirements for the related user interface. Also, the use of different technologies can introduce additional issues from both a usability and safety point of view.

Most of the work that will be presented here was carried out under the aegis of Mefisto Project (http://giove.isti.cnr.it/mefisto.html), in which both CENA (a French institute in charge of studies and research for improving the French ATC systems) and ENAV (the Italian association of controllers) partners provided a number of case studies to test the feasibility of the techniques in real settings. Within such a project, we developed the task model for the existing application, which was useful to grasp activities to support and the roles that perform such tasks, as well as investigate on possible problems and limitations. Then, we model an envisioned application able to support communication using data link, a technology allowing asynchronous exchanges of digital data containing messages coded according to a predefined syntax. The basic goal of the envisioned system is to overcome the limitations of current system, principally the fact that communications are mostly based on VHF-based dialogues, which might quickly become a bottleneck. In fact, at any time only one speaker can broadcast on the radio frequency, thus, if a pilot is communicating with the controller and some urgent requests arrive from other pilots, they might be delayed and, in some cases, the misunderstandings typical of voice communication in an international environment can occur. This new solution should provide controllers with real-time representations of the current traffic even in case of bad atmospheric conditions that limit the visibility from the control tower.

In this chapter we first introduce the reader to the main features of the ATC application domain, both considering the current system and the envisioned one in two different phases of a flight, the en-route and the aerodrome phase. Then, we go along again to the different techniques that have been proposed in Chapters 4-8 for the design, verification and evaluation of interactive safety critical applications, by providing examples respectively in section 9.3.1 (design) 9.3.2 (property verification) 9.3.3 (integration between task models and system models) 9.3.4 (evaluation), where their application to the ATC case study will be illustrated.

9.2. THE ATC DOMAIN

The principal objectives of the air traffic control system are generally stated as the achievement of safe and expeditious flow of traffic through airspace. Safety means that separations standards, for the minimum safe distance between aircraft, should be respected. Expedition means that, as long as safety is preserved, the flow of traffic through the airspace should be maximised.

As you can see in Figure 40, three main phases can be identified within the 'lifecycle' of a flight: the approach phase, the aerodrome phase, and the en-route phase, The *approach* phase is identified as the period of time in which flights are in the immediate vicinity of the airport. This phase raises quite critical issues from the viewpoint of safety because controllers are expected to cope with potential concentration of flights around the airport with strict time constraints that make their work harder and with potential disastrous effects in case of errors. In the *en route* phase the possibility of accidents is usually low, however, as it has been already anticipated, the increasing traffic amount and some limitations of the voice communication have pushed to envision a new system in which *data link* communication are added to VHF. The *aerodrome* phase refers to the period of time in which the flight moves within the aerodrome area, until it reaches (resp.: leaves) the starting point of the runway (so-called 'holding position') and takes off (resp.: lands). Many accidents and incidents have been reported in airports during this phase because are becoming more and more crowded, and there is also a strong need for tools supporting controllers' decision-making.

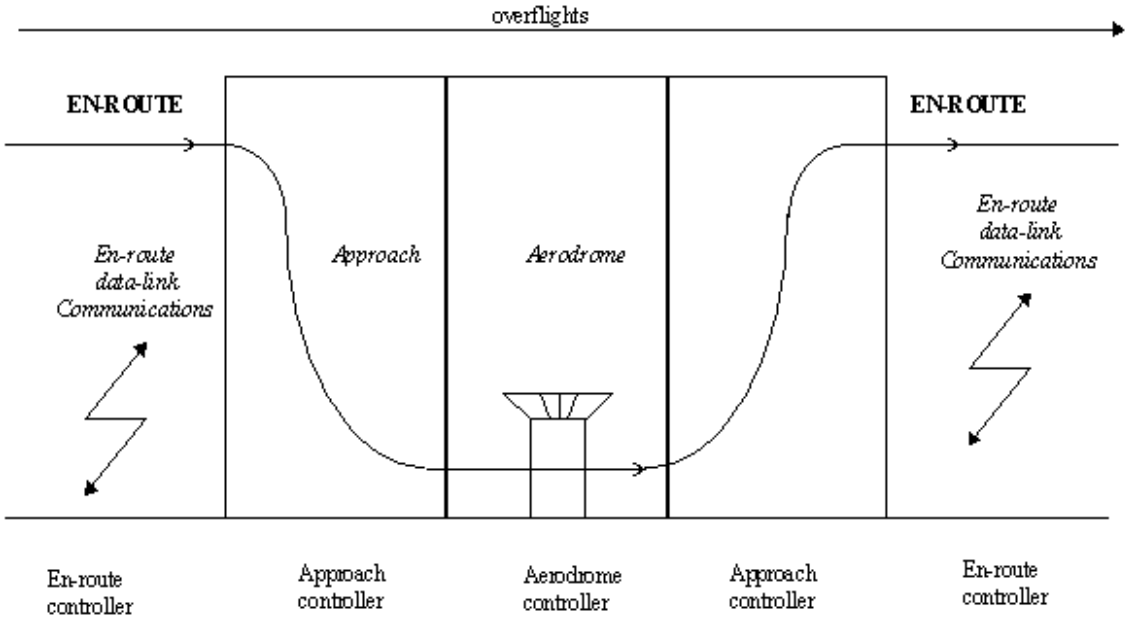

Figure 40. The Main Phases of a Flight

Although all the phases of a flight are safety critical to the aircraft, two main phases have been considered in this chapter, the en-route phase and the aerodrome phase, selected in such a way to allow the reader to have a complete view of the possible usability and safety problems to solve.

They will be described more in depth in the following sections, together with some excerpts of the associated task models. We provide examples taken from the analysis of both the current and the envisioned system for both phases and considering the different roles involved, namely the ground and the tower controller for the aerodrome phase, and the executive and strategic for the en-route one.

9.2.1. The En-route Phase

The VHF-Based System

Civil airspace is partitioned into a number of regions known as *sectors*, by horizontal and vertical divisions. Aircraft within a sector are managed by two air traffic controllers, who work closely together, but having individually different roles and concerns. The *executive* controller is able to contact aircraft using VHF radio, and is directly responsible for making short-term

decisions and maintaining the appropriate separation distance between aircraft. The executive controllers are first contacted by the pilots when the latter one enter the sector; when the pilot leaves the sector, the VHF 'last contact' takes place and the executive controller is in charge of sending to the pilot the frequency for contacting the controller in charge of the new sector. The *strategic* controller, on the other hand, has longer-term concerns and coordinates with strategic controllers of adjacent sectors to agree about flight parameters (particularly the altitude) of aircraft entering and leaving the sector, so that their passage between sectors is carried out in a safe and orderly manner.

The environment of a single en-route working position is shown in Figure 41: controllers are provided with radar screens, on which the location and parameters of aircraft, the airways, and the sector boundaries are displayed. A VHF radio is provided for use by the executive controller (seated on the left in the figure) to communicate with pilots in the sector (this possibility is available also for the strategic only for emergency cases). Telephones allow the strategic controller (seated on the right) to keep in touch with other strategic controllers of neighbouring sectors. A Touch Input Device (TID) allows the ground-based computer system to be updated to reflect changes to flight data (such as the time, flight level and track). Usually, it is the strategic controller who performs these updates.

Figure 41: The Current Enroute Air Traffic Control Position

The controllers on a working position have also some flight paper strips (one strip per aircraft) printed in real time and delivered to a given control position about ten minutes before the flight enters the sector handled by that position. The strip gives general information on the aircraft (identifier, aircraft type) and also information on its planned route.

The Envisioned System

The main difference with respect to the current system is that the envisioned system is that we consider an en-route environment with more electronic support. We thus consider the possibility that both controllers, in different circumstances can send clearances by *data link*, (a technology allowing asynchronous exchanges of digital data coded according to a predefined syntax) and in addition, in this system it is supposed that the information formerly contained in paper flight progress strips can be electronically represented in interactive, *electronic flight strips*, through which it is possible to control and monitor traffic evolutions in the system.

9.2.2. The Aerodrome Phase

The modelling exercise of the envisioned system for managing aircraft in the aerodrome area with data-link support considered a large number of tasks: 89 ground controller's tasks, 72 tower controller's tasks, 63 pilot's tasks, and 73 cooperative tasks (tasks that imply actions from two or more users). Some excerpts of the task models developed for this phase will be detailed later on in this chapter. We show a summary about statistic data on task models in Figure 42.

Figure 42: Statistic Information About Task Models in the Envisioned Aerodrome System

The Current System

In the current system, there are two types of controllers working elbow-to-elbow in the control tower to manage the traffic within the aerodrome: the *ground controller* and the *tower controller*. As in the en-route case study, they communicate through the VHF-based communication system with pilots currently in the sector, by means of telephone with controllers of other centres, and speak directly each other. In addition, they use *paper flight strips* as well to monitor traffic progress. In addition, they use the radar, which allows them to monitor the current traffic situation especially when it is not possible to have a complete overview of the traffic with the naked eye. In addition they have specific duties:

- the *ground controller* has to guide planes from the departure gate until the runway's starting point (holding position) and (for arriving planes) from the end of the runway until the arrival gate.
- *tower controllers* are in charge of maintaining the minimal separation between aircraft then they allocate the access to the runway(s) and decide about take-off and landing.

The Envisioned System

The basic requirements for the envisioned system are the use of *data-link* to support the communication with aircraft and the use of *enriched flight labels* (see Figure 43) to provide information concerning each flight instead of traditional paper strips. The enriched flight labels show permanently just essential information concerning flights but they can interactively enlarge to show additional information and shrink again. They are different depending on the specific controller. When a label is selected, an extended set of information is given including the path assigned to a flight (which can be both textually and graphically represented), the current velocity of the aircraft, and other useful data for the controller (see Figure 43).

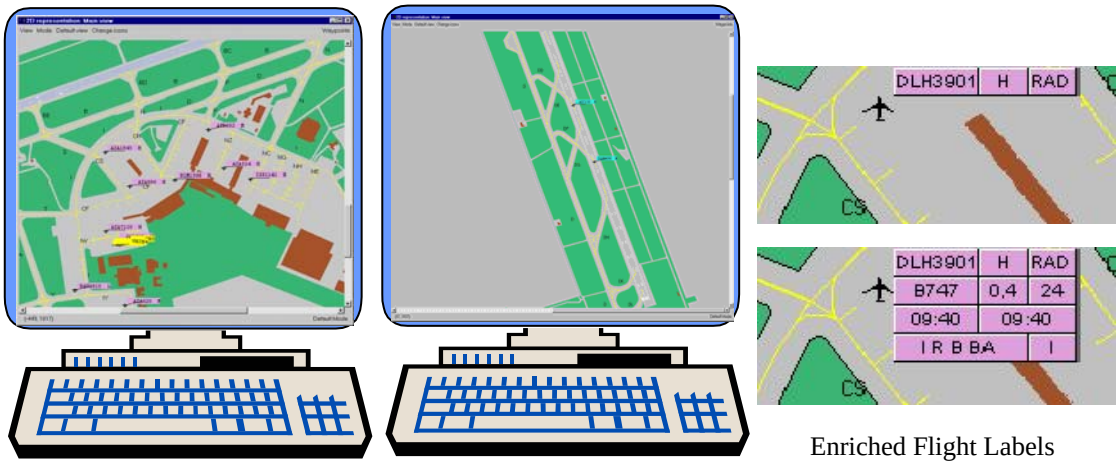

Figure 43: The User Interfaces of the New Aerodrome Prototype Considered

In the aerodrome phase, both controllers have to build and maintain an updated picture of the current state, looking at the taxiways (for the ground) and at the runways (for the tower) and monitoring the tools available on the radar display (see Figure 44 for a screenshot of the prototype we considered for the new environment), which are enriched flight labels, aerodrome map, automatic tools which alert the controllers when some deviation is detected or is near to happen, etc.. Also, both controller receive continuously the data-link requests that arrive and, from time to time they decide which request has to be managed next. If it is a request for the ground controller, s/he has to build and send a path to the pilot, driving and controlling the flight until the pilot gets to the destination point (e.g. holding position for departing pilots). Finally, the controller passes the responsibility of the flight to the next controller. If it is a request for the tower from a departing pilot, the controller has to collect data for the departing flight and find the best answer for the pilot, alternatively, if the request is for landing, the controller has to decide if there are all the safe conditions necessary to make the flight land and then passes the responsibility of the flight to the ground controller, who drives the flight to the apron.

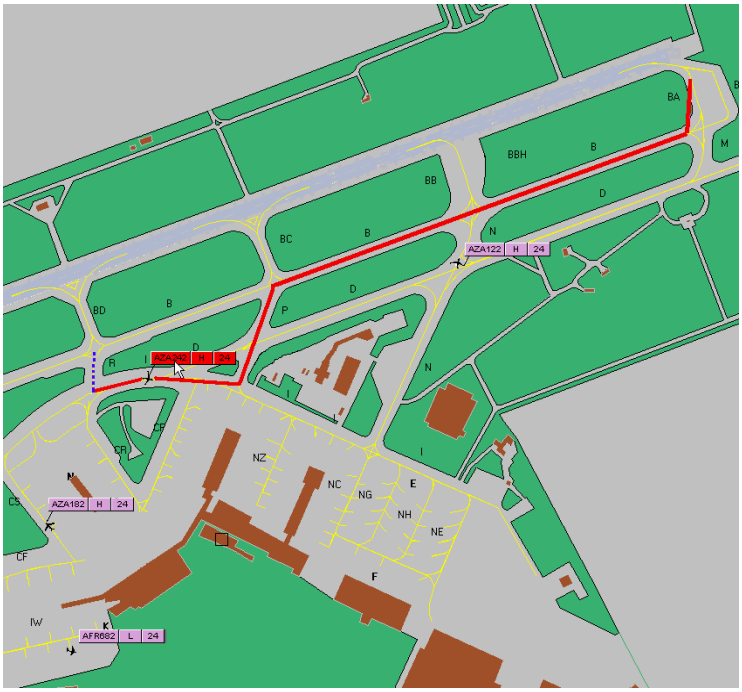

Figure 44. The User Interface of the Prototype Considered (ground controller)

9.3. APPLYING THE APPROACH TO THE CASE STUDY

9.3.1. Designing ATC User Interfaces

We consider the *Build Path* task of the *ground* controller in the *envisioned system*, which describes all the activities necessary to answer a taxi request from a pilot. A taxi request can be presented either by a departing pilot asking for a suitable path to reach the runway from the assigned gate, or by an arriving pilot who has to reach the gate once the aircraft exits the runway. The *Build Path* task is decomposed into two main subtasks (see Figure 45):

- *Build path with automatic suggestions*, which describes the activities performed by the controller to build a suitable path exploiting automatic tools (in this case the system really drives the controller to get the best solution showing the set of possible solutions);
- *Build path without automatic suggestions*, when the controller is more self-confident and builds directly the path (in this case the role of the system is mainly to support controllers by helping their decision-making process).

As far as the subtask *Build path with automatic suggestions* are concerned, it is composed of an application task (*Show calculated ordered paths*) concerning the activity of the system to calculate and show the paths ordered by the parameter that is currently the ordering criterion. This activity can be followed by an iterative subtask (*Ordering paths*) that describes all the activities necessary to allow the user to select different parameters and get the set of possible solutions ordered in a different manner. For example, if the controller selects the parameter "Length", the system will show all the possible paths ordered by this parameter. First, the paths that have the minimum length, and then the others in an increasing order (*Show ordered paths*)

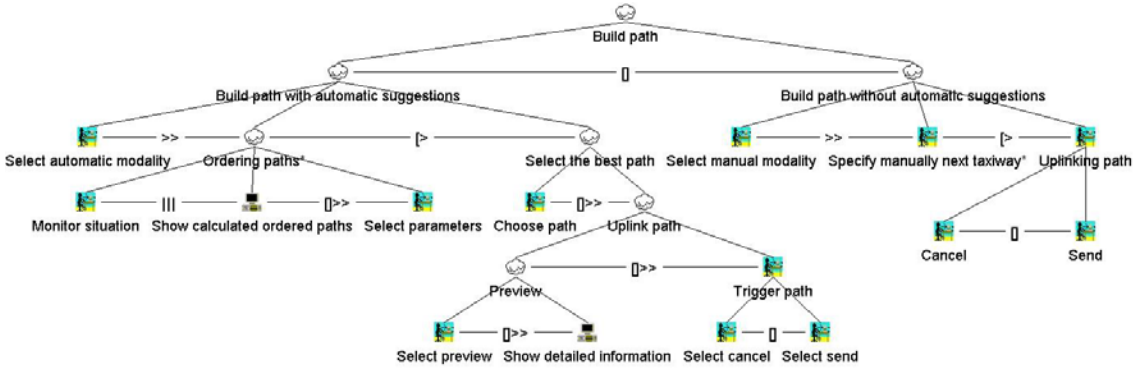

Figure 45: An Excerpt of a task model for the ground controller in the envisioned system

This process can be repeated multiple times (the task is iterative) because the controller can change mind and select another criterion. However, finally, the controller chooses the "best" path and after having (optionally) activated a preview of all the information about this path (*Preview* task), s/he is able to either send the path or cancel the whole process and restart it. In the task model it is modelled by a recursive instantiation of the *Build path* task. The subtask *Build path without automatic suggestions* allows the controller to build the path specifying directly the taxiways and optionally specifying whether and which parameter s/he is interested to know as s/he gradually builds the path. For example, if the controller has selected the parameter "Length", as s/he gradually selects the various segments of the route, the system updates the overall distance in order to help the controller to decide on the suitability of that path option. The dialogue associated to the *Ordering paths* task is composed of two logical parts, the first one dedicated to the selection of the parameters, and the second one where the set of paths are displayed according to the chosen criterion.

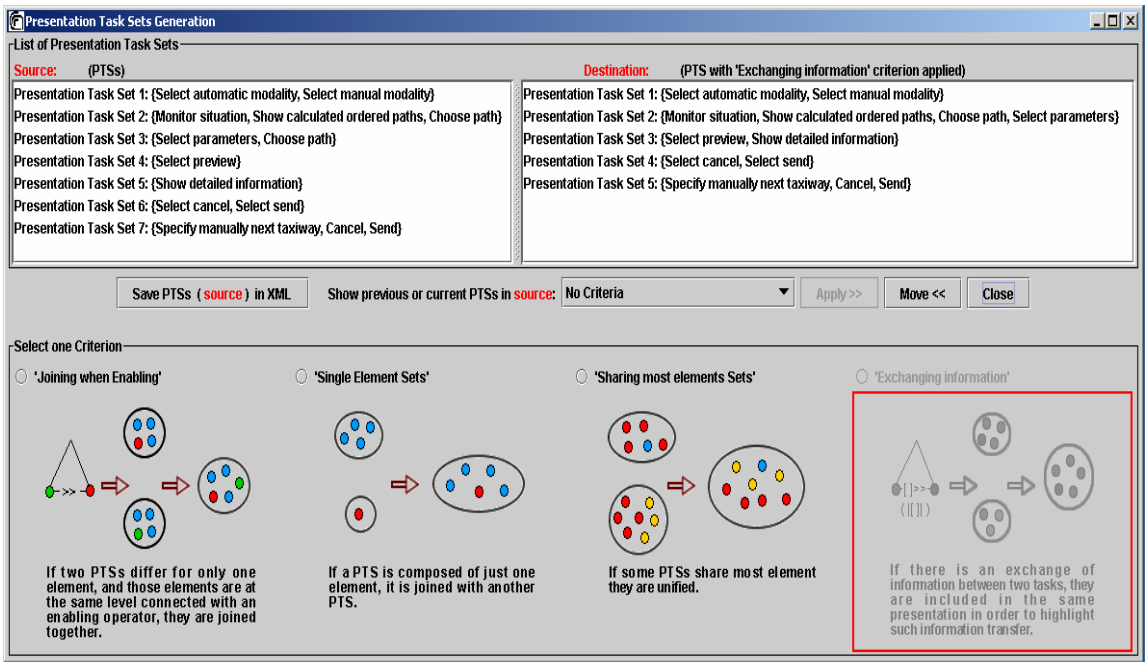

Figure 46: Example of Calculation of Presentation Task Sets

In Figure 46 the PTS automatically calculated for the task model described in Figure 45 have been displayed. In particular, in the left side of the figure the PTS automatically derived from the model are illustrated, while, in the right side the PTS derived by applying the *Exchanging Information* criterion are visualised. By analysing the list in the left hand part in comparison with the related task model, the reader should grasp the main points of how the semantics of the temporal operators affects the generation of Presentation Task Sets. For instance, analysing Presentation Task Set 6 we can note that such set is composed of two tasks (*Select cancel*, *Select send)* which are connected through a choice operator, so they appear in the same set. In fact, in order to carry out the selection both of them should be enabled over the same period of time.

Furthermore, as we said before, the enabling operator identifies the 'border line' between different sets, because it requires that the performance of the second task should be enabled just after the completion of the first task, so the two tasks should not belong to the same set. This situation is exemplified in PTS4 and PTS5 of Figure 46. The effects of translation the disabling operator can be seen in PTS7, where the tasks *Cancel* and *Send* appear together as they are linked through a choice, and the *Select manually next taxiway* is included as well because the semantics of the disabling operator requires that the tasks that appears at the right side of a disabling operator should be enabled over the whole time interval in which the tasks that could be disabled are enabled. This is because the interruption could occur any time during the performance of the concerned tasks.

In the right part of the window it is possible to see the result of combining together two or more PTS by applying a criterion according to which all the tasks that exchange information each other might be included in the same set in order to highlight this transfer of information. As you can see, after this application the set is shrunk from seven to five sets.

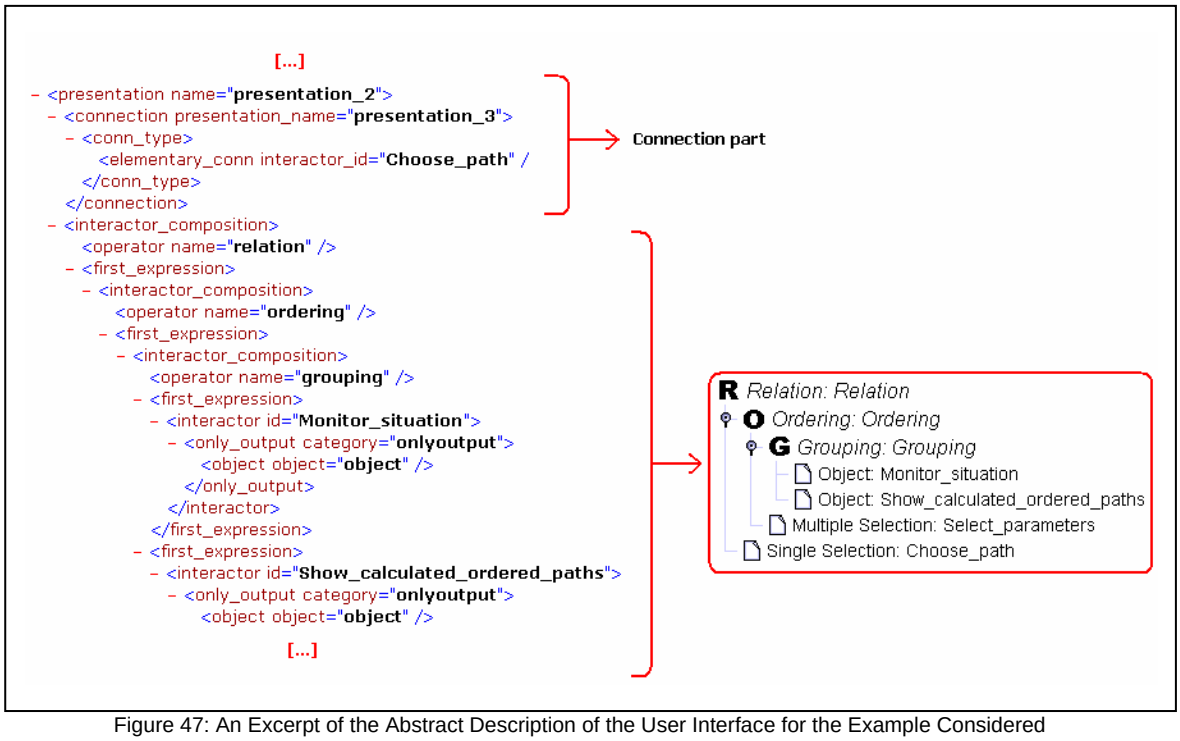

Figure 47: An Excerpt of the Abstract Description of the User Interface for the Example Considered

In Figure 47 we have provided a small excerpt of the user interface description that has been derived from the task model specification shown in Figure 45, and after applying the aforementioned heuristic that allows us to obtain the set of PTS shown in the right part of Figure 46. As you can see, in the related PTS we have four tasks, and then in the related abstract presentation we have four interactors that have been appropriately derived following the approach outlined in Chapter 5. As you can see, the interactor associated to the "Choose path" task is aimed at two different goals, the first one is allowing the user to perform a selection between the different paths (as you can see it is associated to a single selection task), the second one is to enable a change of presentation in the user interface. As you can see from the XML-based excerpt in the left hand part of Figure 47, when the user selects the path the dialogue is supposed to move from the current presentation to the next one, which is presentation_3. In particular, we focused on the presentation that is derived from the PTS2, which is composed of the following tasks: *Select parameters*, *Show calculated ordered paths*, *Choose path*, *Monitor situation*. As you can see, between the interactor associated to the selection of parameters and the interactor associated to the visualisation of the paths ordered according to such parameters there is an ordering relationship meaning that the system should initially allow the first activity, and then highlight the results afterwards. As you can see from the task model, this activity could occur more than one time, than it useful to have the related interactors grouped in the same presentation, in order to show the flow of information exchanging between these interactors.

A possible implementation for this ideas is shown in Figure 48, The first panel is associated to the parameters which the controller could be interested to know, e.g. the calculated length of the path, the number of foreseen runway crossings, how much time the travel will take (supposing a standard velocity on the taxiways), and so on.

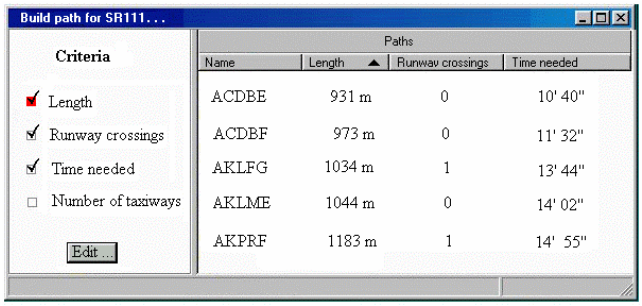

Figure 48: A possible Implementation for the Presentation 2

The user interface should allow the user to select a list of parameters from a pre-defined set (allowing multiple selections) and mark one of these (single choice) as the sorting criterion. A possible implementation is shown in Figure 48: controllers can mark which parameters are relevant to them (a "√" is associated to each selected parameter) or they can select a different set of parameters ("Edit" button). However, only one parameter can be selected as the ordering criterion, which is highlighted by a different presentation technique in the user interface. There is also the possibility to choose the ordering criterion on the right side of the window, by selecting a specific column, which becomes in such a way the current ordering parameter.

Other considerations have to be done about the type and cardinality of data, being the ultimate choice about the specific presentation that has to be used (especially that exploiting multimedia features) dependent on the integrated consideration of all those aspects. For example, being the path a spatial data, using some graphical technique is a good way to present it. This can be confusing when there are many paths that have to be displayed at the same time. In this case, a good solution is to provide more than one type of presentation, for example an immediate textual path, and the possibility of having a graphical presentation on request. For example selecting a textual representation of a path, then the path is graphically highlighted in a separate window as shown in Figure 49.

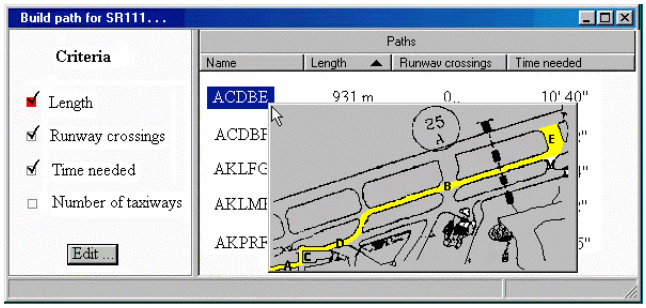

Figure 49: An Example of Implementation for the Preview

9.3.2. Verifying Properties

In this section we consider an excerpt of a CTT task model specification describing the main activities performed by the *tower* controller in the new interactive prototype application for the *envisioned* aerodrome, which uses data link technology for communications. In Figure 50 we show the related task model that we consider in this section for verification purposes.

The Task Model

As you can see from Figure 50, the task of handling the traffic (*Handling Traffic*) is constituted by two main activities: check about possible conflicts currently occurring (*CheckConflicts*), and managing the communications with the pilots about clearances (*ManagingClearance*). *CheckConflicts* decomposes into two concurrently performed sub-activities, depending on whether the controller is currently observing the aerodrome map (*CheckConflictsInMap*) or the controller is currently checking the aerodrome state directly

through the control tower window (*CheckConflictsThroughWindow*). In the first case, the activity considers the possibility of activitating the zooming functionality in (*ZoomIn* task) or out (*ZoomOut*) the aerodrome area. Such zooming activities are iterative and concurrently performed (because the controller can decide to zoom in or out the map a certain number of times and in whatever order) and both can be possibly deactivated when the controller decides to return to the default view of the aerodrome area (*ReturnToDefaultView*). Such a view allows the controller to have the complete picture of the current situation in the whole aerodrome area.

When the controllers decides that the current level of detail is the right one for their goals, they are able to correctly perform the activity of monitoring the flights currently shown in the map (*MonitorFlightsOnMap*) and, as soon as they detect a problem they have to find a suitable strategy to resolve the problem and then move on to actually solving it (*SolveProblem*). Also, the controller can decide to check deviations directly looking through the tower window (*CheckConflictsThroughWindow*), which involves performing the activities requested to manage conflicts (note in the tree the occurrence of *CheckDeviation* task).

The *ManagingClearance* task consists of concurrently managing departures (*ManagingTakeOff*) and arrivals (*ManagingLanding*). Both *ManagingTakeOff* and *ManagingLanding* tasks are iterative as they are continuously performed by the controller. More specifically, if a take-off request is received (*ReceiveTakeOffRequest),* after a certain period of time the controller will allow the airplane to depart (*SendOkTakeOff*), and this temporal relationship is modelled by the enabling operator ">>". On the other hand, after a request to land is received by the controller (*ReceiveLandingRequest*), the possible replies that can be enabled are either a negative answer (*SendGoAround*) or an affirmative one (*SendOkToLand*), depending on the current traffic situation.

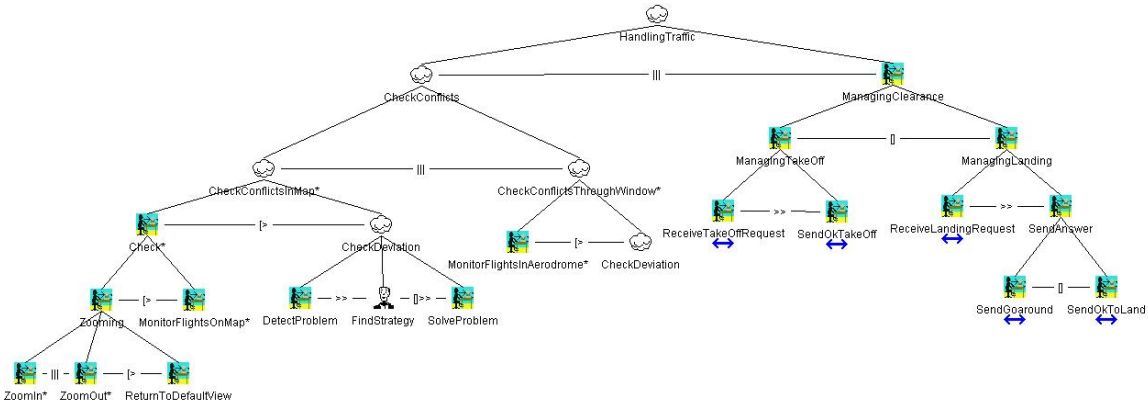

Figure 50. An Excerpt of a ConcurTaskTrees Specification (Tower Controller, Envisioned System)

The Application of the Method

As we indicated in Chapter 6, in order to verify properties on the considered task model, the first step of our approach is to translate the CTT specification into a LOTOS specification, which is automatically performed within the CTTE. The second one is to specify a property that has to be checked against this model. For example, if the user wants to verify if it is possible for the Pilot to execute the landing procedure (*ExecuteLanding*, the name of the ending task*)* once the Tower has received a landing request *(ReceiveLandingRequest*) then he has to select the *Existential Relative Reachability* item in the property list (Figure 51).

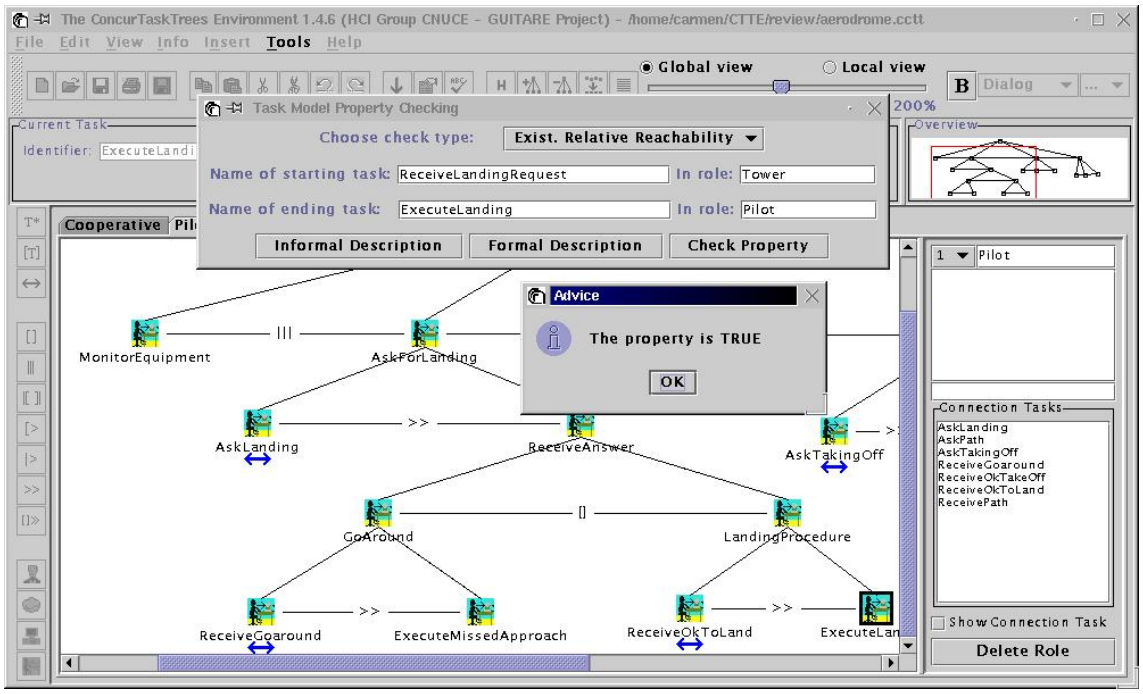

Figure 51: An Example of Calculation of a Propriety

This property means that we want to verify if after the execution of a fixed task it is possible to reach the execution of another task in (at least) one possible next temporal evolution of the system. In order to do it, the user has to select graphically the tasks involved in the property within the model. In case of relative reachability, two tasks have to be selected: the left button of mouse allows designers to specify the first task and the right-button the second task. As a consequence, the associated fields are automatically filled, together with the correspondent roles involved (respectively Tower and Pilot in our example). As you can see from Figure 51, the result in this case is true because in the modelled system at least one possible execution exists such that after having performed the first task allows to reach and execute the second task. A more significant example of how to use the diagnostics produced by the model-checker is when a property is not verified in the system. In this case the diagnostic yields the set of the possible transition sequences that do not satisfy the property. Such sequences, which are the actual counter-examples of the property, are mapped onto a CTT scenario.

The new example involves the same two tasks as before in order to highlight the difference between the Existential Relative Reachability and the Universal Relative Reachability. Beforehand, we verified that there is at least one possible way to reach the execution of the *ExecuteLanding* task after having executed the *ReceiveLandingRequest*. Now we want to verify whether in all the possible temporal evolutions of the system after the performance of the *ReceiveLandingRequest* it is possible to execute the *ExecuteLanding* task. This property is stronger than the previous one and is false in the system specified. In fact —for safety reasons— it is not taken for granted that whenever there is a pilot's request for landing, an affirmative answer will be sent by the controllers: they must be free to decide whether or not it is safe to let the aircraft land in that particular moment.

In this case, the tool shows (one) execution trace provided as a counter-example for this property. It is possible to map the sequence of actions defining the counter-example given by the model checker to the corresponding tasks in the ConcurTaskTrees task model (see Figure 52). A possible counter-example for the property is that after the tower receives the landing request, the controller decides to reply by sending the order of "going around", so that the pilot can execute the missed approach procedure and wait for a safer moment to land. This execution is highlighted by the list of tasks that compose the scenario associated with the counter-example: although the *ReceiveLandingRequest* task appears in the scenario, it terminates

without making the aircraft land (because the aircraft executes a “Missed approach” procedure). Apart from being more readable and intuitive than pure executable traces of low-level actions, scenarios allow users to get executable examples of sequences of basic tasks that can be simulated and followed within the CTT environment. This can be useful to analyse the alternatives that could be considered during the performance of the scenario.

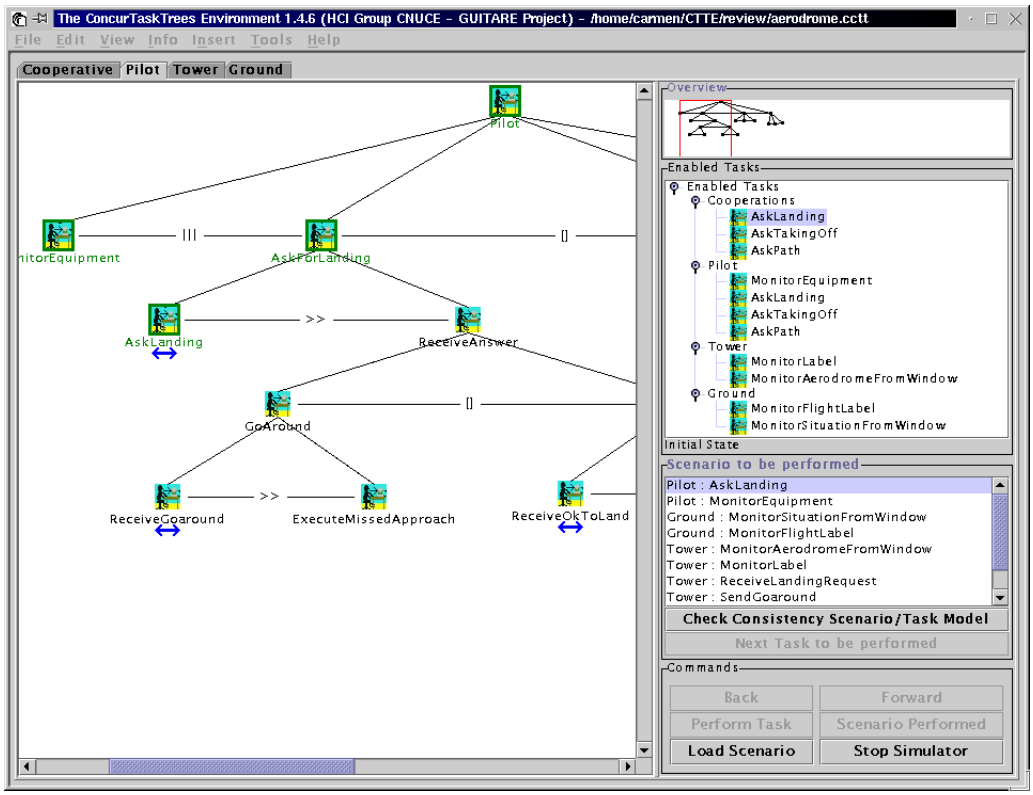

Figure 52. A Model-Checker Counter-Example Translated onto a CTT Scenario Ready for Interactive Simulation

During our verification exercises we also obtained some interesting results that allowed us to suggest modifications to the prototype in order to obtain an improved one -in terms of safety and usability. An example of a property which was useful to deduce suggestions for improving the current prototype was a property involving the currently implemented zooming activities available in the prototype to the controllers. In fact, as we previously described, in the prototype it was possible for the controller to zoom in or out of the aerodrome map in order to have a more detailed view of some parts of the aerodrome, together with the possibility of having a button to return to the default view. However, if the controller does not return to the default view it is possible that some conflicts occur in the part of the airport currently out of the controller’s view and are not detected, because the controller has just zoomed in a different part of the aerodrome map. So, our property aimed at verifying if it is true that, after having activated a zooming function at a certain point, sooner or later during every next evolution of the system the controllers will return to the default view, which guarantees that no aerodrome area is neglected. If we use the templates previously introduced, this property can be easily expressed in the following way:

```
Universal Relative Reachability(ZoomIn,  ReturnToDefaultView)
```

The property aims at verifying whether it is true that once one *ZoomIn* action is performed, then for all the possible subsequent temporal evolutions, it is true that the controller will sooner or later perform a *Return ToDefaultView* action, which provides the complete picture of the whole aerodrome area at a glance. It is worth noting that the same property could also have been checked for the *ZoomOut* action.

When we verified this property, it did not hold in the prototype considered. Such a finding allowed us to suggest the possibility of improving the prototype by adding some mechanism able to guarantee that the controller does not neglect any part of the aerodrome. During other exercises we were able to draw also other interesting results from the verification process, which were the starting point for other useful discussions within the project.

9.3.3. Integrating Task Models and System Models

In order to exemplify the approach illustrated in Chapter 7 about integrating information of task models with that incorporated within system models, we consider an example from the en-route phase with the support of data-link technologies, focussing in particular on the activities related to the controller's activity of transferring a plane to an adjacent sector.

A representation of the radar image available to the en-route executive controller is shown in Figure 53. On the radar image each plane is represented by a graphical element providing air traffic controllers with information for handling air traffic in a sector. In the simplified version of the radar image we are considering, each graphical representation of a plane is made up of three components: a label (providing precise information about the plane such as ID, speed, cleared flight level, …), a dot (representing the current position of the plane in the sector) and a speed vector (a graphical line from the dot which represent the envisioned position of the plane in 3 minutes). An air traffic control simulator is in charge of reproducing the arrival of new planes in the sector while in reality they would be instantiated on the user interface by calls from the functional core of the application processing information provided by physical radars.

Initially the radar image is empty. Each time a new plane is randomly generated it is graphically represented on the radar image. It is possible for the user to select planes by clicking on its graphical representation. Clicking on the flight representation will change its state to the 'Assume' state meaning that the air traffic controller is now in charge of the plane. Assuming the plane changes its graphical representation as it can be seen on the right-hand side of Figure 53. Once a plane is assumed, the controller can send clearances to this plane. When the plane has been taken on, the button 

is enabled (see plane 1123 on the right-hand side of Figure 53). Clicking on this button opens a menu allowing the controller selecting the new value for the frequency.

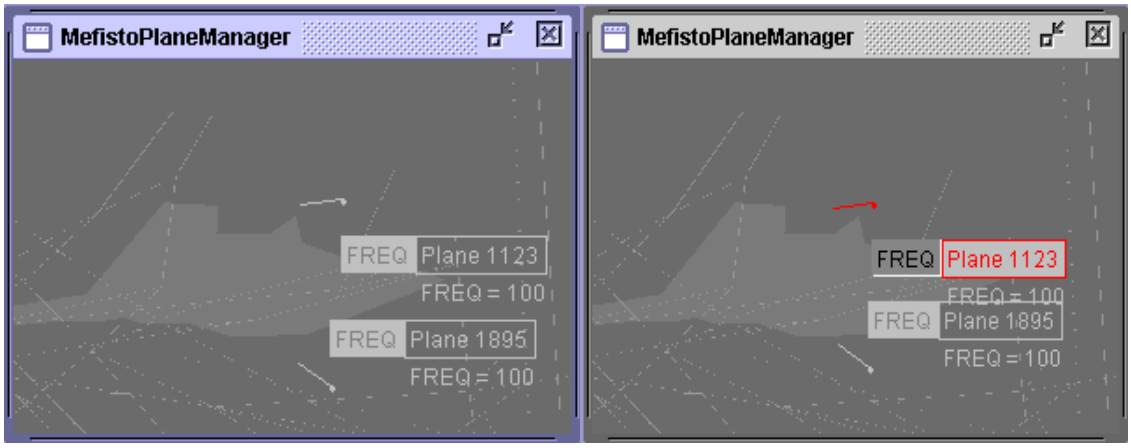

Figure 53. A Screen Shot of the Radar Screen with Planes (Right-Hand Side, Once the Plane 1123 is Assumed)

The Task Model

If we consider more in depth the activity of controlling a plane (see the related task model in Figure 54) is an iterative task (see the iterative operator * beside the root task *Control a plane*) which consists of either assuming a plane (*Assume a plane* task) or giving clearance to the plane (*Give clearance* task). Those two activities are mutually exclusive, as you can see from the choice operator []. The activity of assuming a plane is composed of deciding which plane has to be assumed (*Select a plane* task, the associated icon emphasizes the cognitive

nature of this *user* task). Once this activity has been performed, it is possible to select the button related to the plan (see the Enabling operator with information passing []>>, which highlights that only after the first activity has been carried out and information has been delivered to the second task, the latter can be performed).

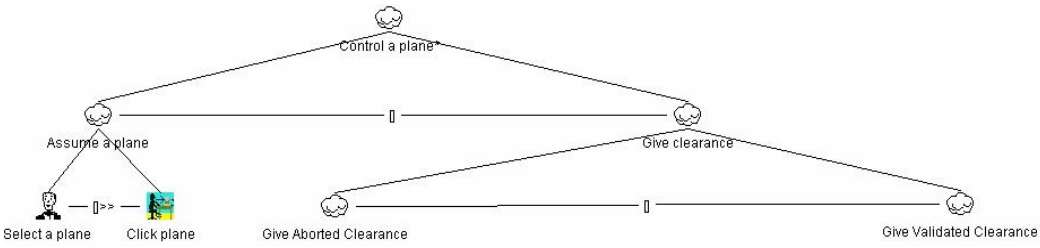

Figure 54. The Task Model of the Example

Then, *Click plane* task requires an explicit action of the controller on an element of the user interface so it belongs to the category of *interaction* tasks. The *Give clearance* task is composed of two different activities: *Give Aborted Clearance* and *Give Validated Clearance*, depending on whether the clearance has been aborted or not. Each of these two activities is a high-level one, whose performance cannot be entirely allocated either to the application alone, or to the user alone, or to an interaction between the user and the system, so this is expressed by an *abstract* tasks. The specification of each of these two tasks is described in Figure 55.

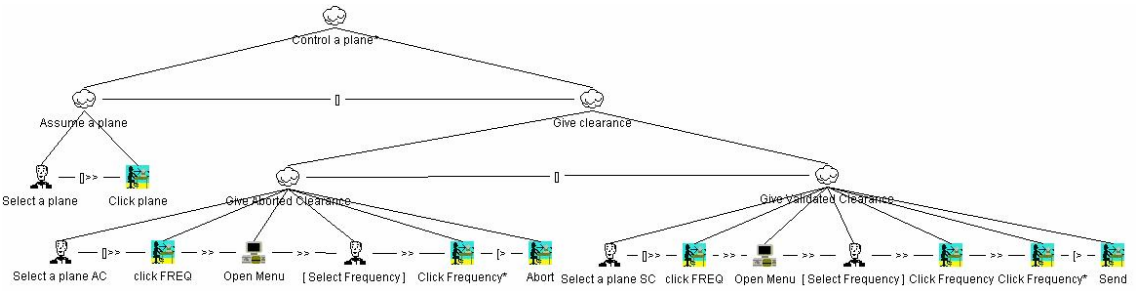

Figure 55. The Detailed Task Model of the Case Study

The *Give Aborted Clearance* task is composed of the controller's cognitive activity of selecting a plane (*Select a plane AC*), then they select the button related to the frequency (*Click FREQ*). This triggers opening the associated menu on the controller's user interface (Open Menu, note the icon associated to the category of the application tasks), then the controller can think about a specific frequency (in the task model the possibility of performing or not this task is expressed by the option operator represented by squared brackets [T], see the *Select Frequency* task). Then, controllers choose the appropriate value of frequency within the user interface (*Click Frequency* task, which can be performed more than one time, as you can see from the iterative operator *) until they decide to interrupt the entire activity (see the Disabling operator "[>" which refers to the possibility for the second task to disable the first one), by selecting the related object in the user interface (*Abort* task). In case of a clearance that is sent to the pilot (*Give Validated Clearance*), the sequence of actions is mainly the same, apart from the last one (*Send* task), with which the controller sends the clearance to the pilot.

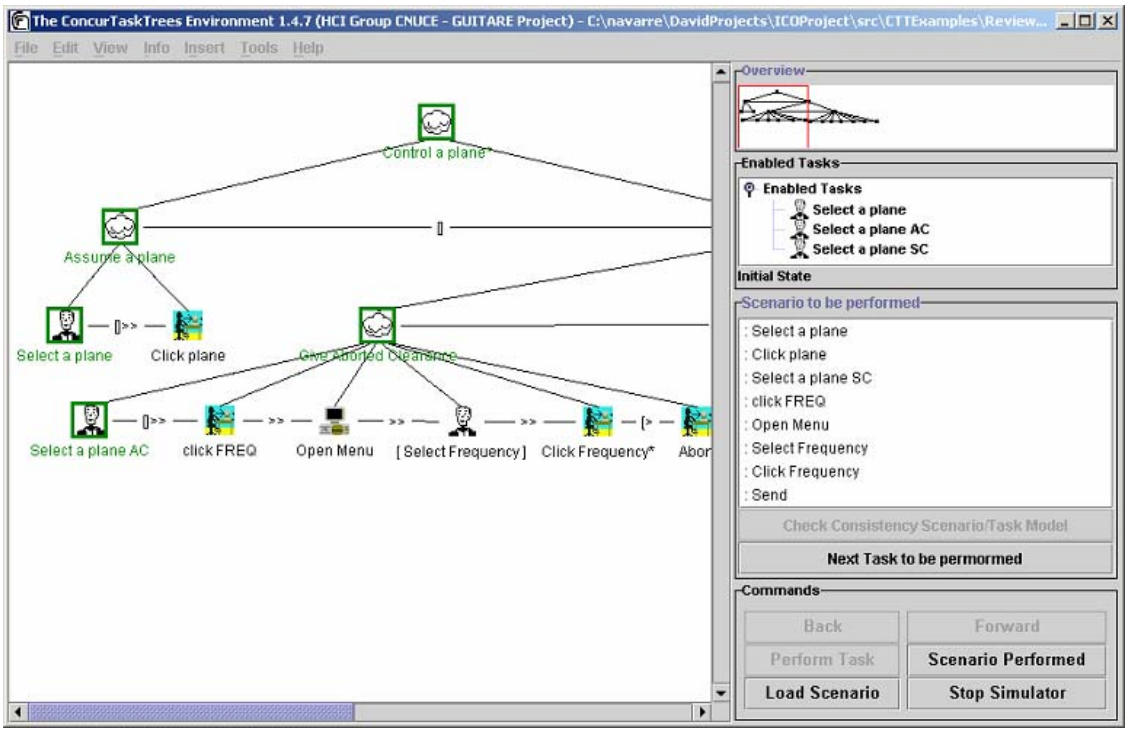

Figure 56. CTTE for Extracting Scenarios

In Figure 56 the simulator provided in the CTT Environment is shown. When this tool is active, in the left-hand part of the window it is possible to highlight (by means of a different colour) all the tasks that are enabled in the current moment of the simulation. This mainly involves all the tasks that can be performed at a specific time, depending on the tasks that have been previously carried out. In addition, it is possible to load a previously created scenario, whose composing tasks will appear in the "Scenario to be performed" list from where it is possible to simulate its performance again.

As an example of scenario extracted from the task model of Figure 55 the following trace of low-level tasks has been considered (this scenario has been generated using CTTE and you can see it displayed on the right-hand side of Figure 56):

- first the controller selects one of the planes not yet assumed (this is the user task “Select a plane”;
- then the controller clicks on this plane to assume it (interaction task “Click plane”)
- then the controller decides to change the current frequency of one of the flight assumed (user task “Select a plane SC”)
- then the controller clicks on the label FREQ to open the data-link menu (interaction task “click FREQ”)
- then the controller opens the menu of frequency for this plane (interaction task, “Open Menu”)
- then the controller selects (in his /her head) a new frequency for this plane (user task, “Select Frequency”)
- then the controller clicks on one of the available frequencies for this plane (interaction task, “Click Frequency”)
- then the controller clicks on the SEND button to send the new frequency to the aircraft (interaction task, “Send”)

The performance of this scenario on the system model will be detailed later on in this section.

The ICO System Model

In this section we only present a subset of the classes and objects of the case study specification. Such subset is composed of three cooperating classes: MefistoPlaneManager, MefistoPlane and MefistoMenu. Their components will be detailed later on.

The class MefistoPlaneManager

The class MefistoPlaneManager is the class in charge of handling the set of planes in a sector. Each time a new plane arrives in the sector the MefistoPlaneManager instantiates it from the class MefistoPlanes (which will be detailed in the next section). During the execution this class will only have one instance. The IDL description shown in Figure 57 highlights that the class offers three services dealing with the managing of the planes in a sector: adding a plane, terminating a plane (when it leaves a sector) and closing the menu of a plane.

```
interface MefistoPlaneManager {
void closePlane(in MefistoPlane p);
void terminatePlane(in MefistoPlane p);
void addPlane(in MefistoPlane p);
};
```

Figure 57. IDL Description of the Class MefistoPlaneManager

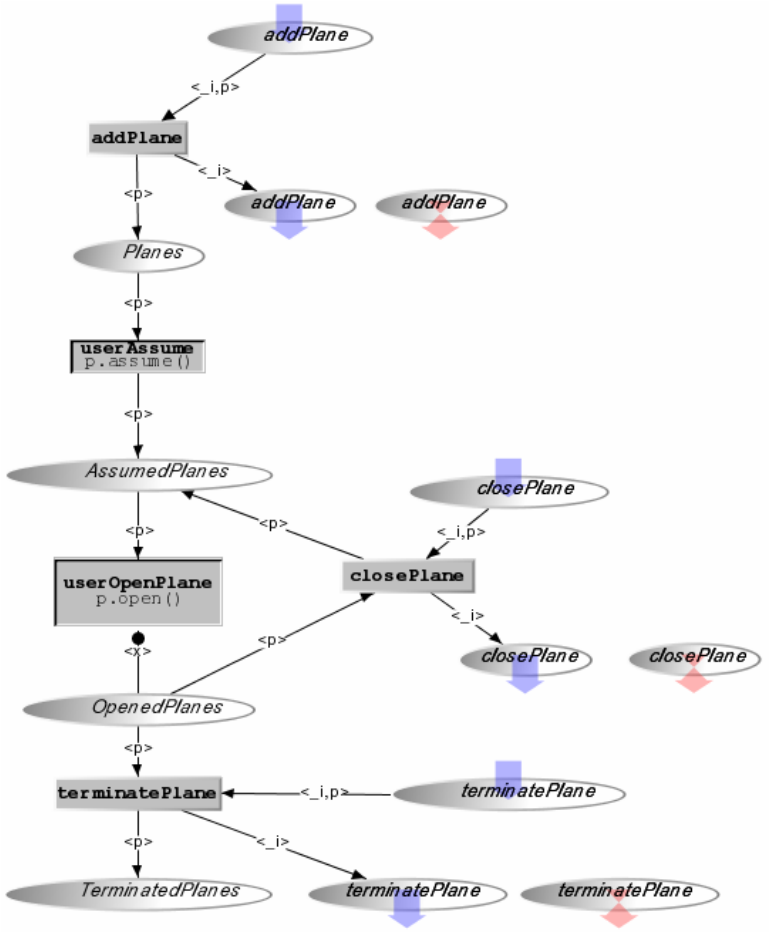

Figure 58. ObCS Description of the Class MefistoPlaneManager

Figure 58 presents the behaviour of this, namely the state of the class and, according to the current state, what the services available to the other objects of the application are. The

transition UserOpenPlane has an input arc from place *AssumedPlanes* meaning that a controller can only open a menu on a plane that has previously been assumed. The inhibitor arc between that transition and the place OpenedPlanes states that only one plane at a time can have the data-link menu opened.

Widget		**Event**	**Service**
Place	**Type**		
Planes	Plane	LabeClick	userAssume
AssumedPlanes	Plane	ButtonClick	userOpenMenu

Table 8. The Activation Function of the Class MefistoPlaneManager

ObCS Element	**Feature**	**Rendering method**
Place Planes	token <p> entered	p.show()

Table 9. Rendering Function of the MefistoPlaneManager

Table 8 and Table 9 describe the presentation part of the ICO MefistoPlaneManager. From the rendering function it can be seen that this class only triggers rendering through the class MefistoPlane, because each time a new token enters in the place Planes the graphical function Show is triggered on the corresponding plane.

The class MefistoPlane

```
interface MefistoPlane {
void open();
void close();
void assume();
void validate(in short x);
};
```

Figure 59. IDL Description of the Class MefistoPlane

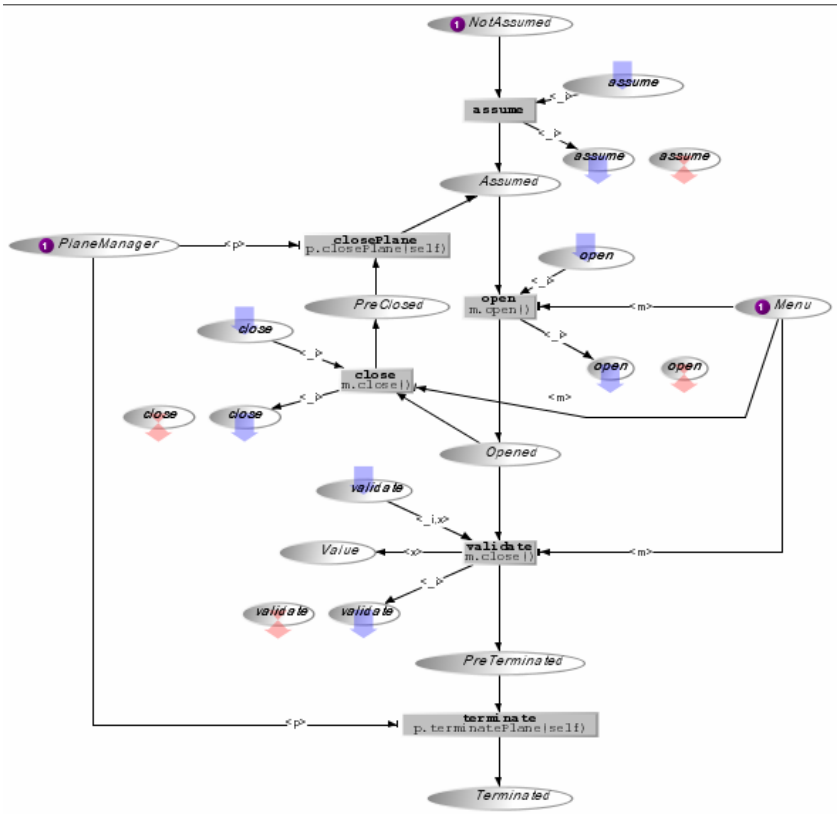

Figure 60. ObCS of the Class MefistoPlane

The class MefistoPlane is also an ICO class. Graphical information is added with respect to the class MefistoPlaneManager in order to describe how the plane is rendered on the screen. This information is given on Figure 61, while Figure 59 gives the IDL description, Figure 60 describes the behaviour of MefistoPlane and Table 10 presents the rendering function. It is interesting to notice that this class does not feature an activation function. This is due to the fact that all the user interaction on a plane takes place through the MefistoPlaneManager class.

```
public class WidgetPlane {
//Attributes
   //A button to open the menu for the change of frequency
   Button freqButton ;
   //A label to display the name of the plane
   Label label;
//Rendering methods
   void show () {//show plane
   }
   void showAssumed () {//show plane as assumed
   }
   void showOpened () {//show plane as opened
         }
   void showTerminated () {//show plane as terminated
         }
   void setFreq(short x) {//show the new frequency
   }
}
```

Figure 61. The Presentation Part of the Class MefistoPlane

ObCS Element	Feature	Rendering method
Place Assumed	token entered	showAssumed
Place Opened	token entered	showOpened
Place Terminated	token entered	showTerminated
Place Value	token <x> entered	setFreq(x)

Table 10. The Rendering Function of the Class MefistoPlane

The class MefistoMenu

This class is in charge of the interaction taking place through the data-link menu that is opened by clicking on the button FREQ on the plane label.

```
interface MefistoMenu {
void open();
void close();
void send();
void setValue(in short x);
};
```

Figure 62. IDL Description of the Class MefistoMenu

Figure 62 provides the set of services offered to the other objects of the application, namely open, close, send and set a value; Figure 63 describes its behaviour.

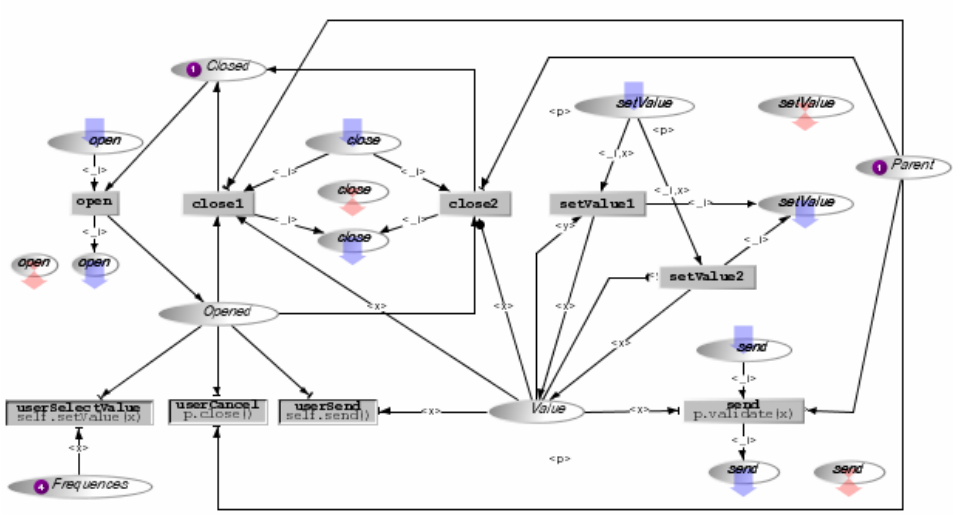

Figure 63. ObCS of the Class MefistoMenu

```
public class WidgetMenu {
//Attributes
   //Button to validate or cancel the current choice for frequency
   Button sendButton, cancelButton ;
   //A comboBox to show the set of possible frequency
   ComboBox freqComboBox;
//Rendering methods
   void show () {//show menu as opened
   }
   void hide () {//hide menu
   }
}
```

Figure 64. The Presentation Part of the Class MefistoMenu

Figure 64, Table 11, and Table 12 give the presentation part of the class MefistoPlane.

Widget	Event	Service
sendButton	actionPerformed	userSend
abortButton	actionPerformed	userCancel
freqComboBox	Select	userSelectValue

Table 11. The Activation Function of the Class MefistoMenu

ObCS Element	Feature	Rendering Method
Place Opened	Token entered	Show()
Place closed	Token entered	Hide()

Table 12: Rendering Function of the Class MefistoMenu

For sake of simplicity we do not put the code of the functions given in Figure 64 and in Figure 61 for describing precisely the graphical behaviour of the classes.

Application to the Case Study

This section presents the application of the integration framework presented in Chapter 7 to the Air Traffic Control case study presented in Section 9.3.3.

Figure 65 presents the correspondence editor introduced in Section 7.3 The left-hand side of the window contains the task model that has been introduced previously and loaded into the correspondence editor. In this panel the set of interactive tasks are displayed. On the right-hand side of Figure 65 the panel “Set of User Services” displays the set of user services in the ICO specification that has been loaded. Here again it is possible to load several ICOs. The set of user services of each ICO appears in a separate tab widget. The lower part of the window in Figure 65 lists the set of associations that have been created when all the user services loaded in the ICOs have been associated with all the interactive tasks loaded in the “Task Tree” panel. Then, the “Launch Scenario Player” button is available. Clicking on this button opens the window presented in Figure 66 corresponding to the scenario player. This tool allows for loading a scenario (produced using CTTE tool) and executing it in Petshop. The scenario can thus be used to replace user interactions that would normally drive the execution of the ICO specification.

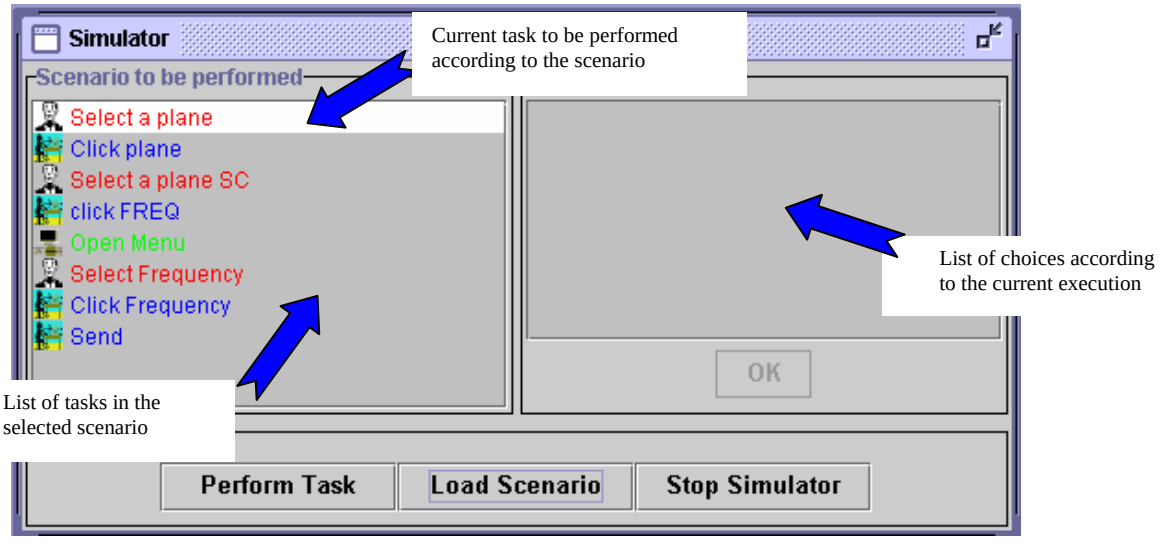

Figure 65: The Scenario Player

The left-hand side of Figure 65 presents the set of actions in the selected scenario. The first line in the scenario represents the current task in the scenario, which is “Select a plane” and is a user task i.e. the task is performed entirely by the user without interacting with the system. Clicking on the “Perform Task” button triggers the task and next task in the scenario becomes the current task. Figure 66 shows the scenario player in use. The right-hand side of the figure shows the execution of the ICOs specification with the two main components: the Air Traffic Control application with the radar screen and the ATC simulator allowing for test purpose to add planes in the sector.

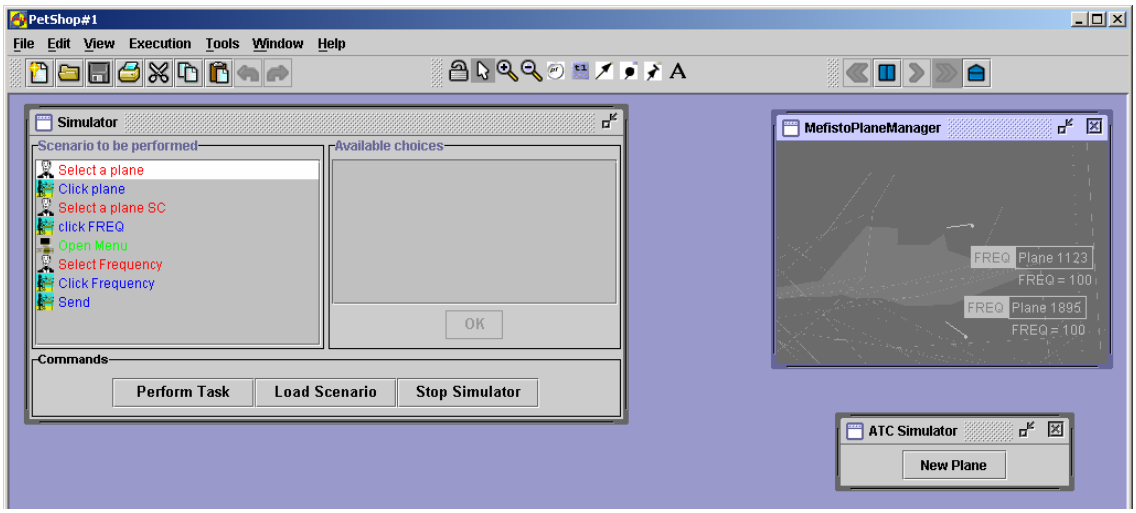

Figure 66. Execution of the Scenario of the System

Some tasks of interactive or application category require runtime information to be performed. For instance this is the case of interactive task “Click plane” that corresponds to the user’s action of clicking on a plane. Of course the click can only occur on one of the “current” planes in the sector and thus, the identification number cannot be known at design time and thus cannot be represented in the task model.

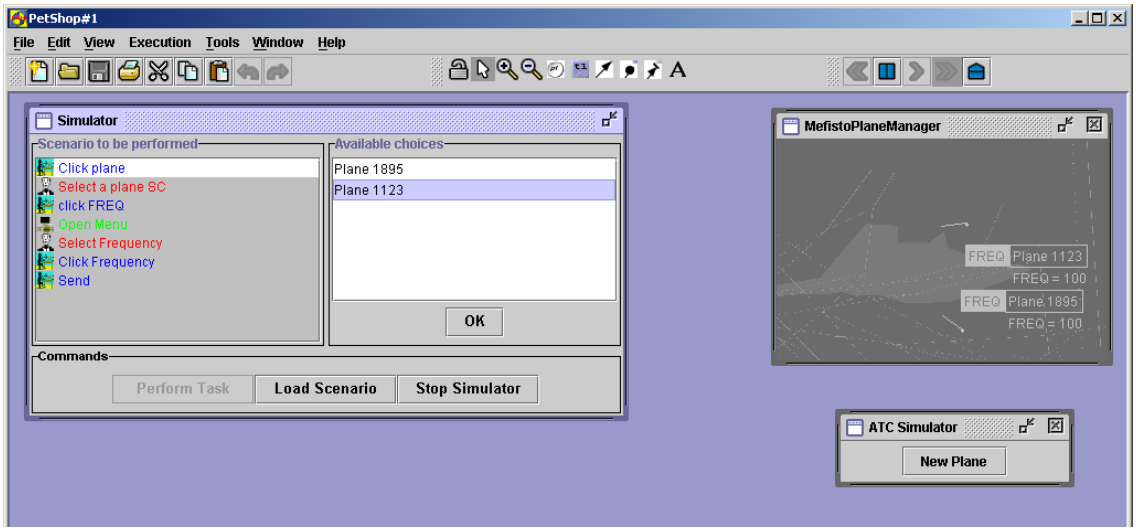

Figure 67. Interaction between Scenario and Current Execution: Planes IDs are Selected at Runtime

Figure 67 provides an example of this aspect. Triggering the action “Click plane” in the task model requires a parameter i.e. a plane identifier. As this interactive task has been related to the user service “userAssume” (in the correspondence editor) the triggering of this task starts off the corresponding user service. However, the triggering of this service requires one of the values in the input place of the transition userAssume in the ObCS of the class MefistoPlaneManager (see Figure 58) i.e. one of the objects planes in the place Planes. In order to provide those values

to the scenario player the set of all the objects in the place Planes is displayed on the right-hand side of the scenario player (see Figure 67).

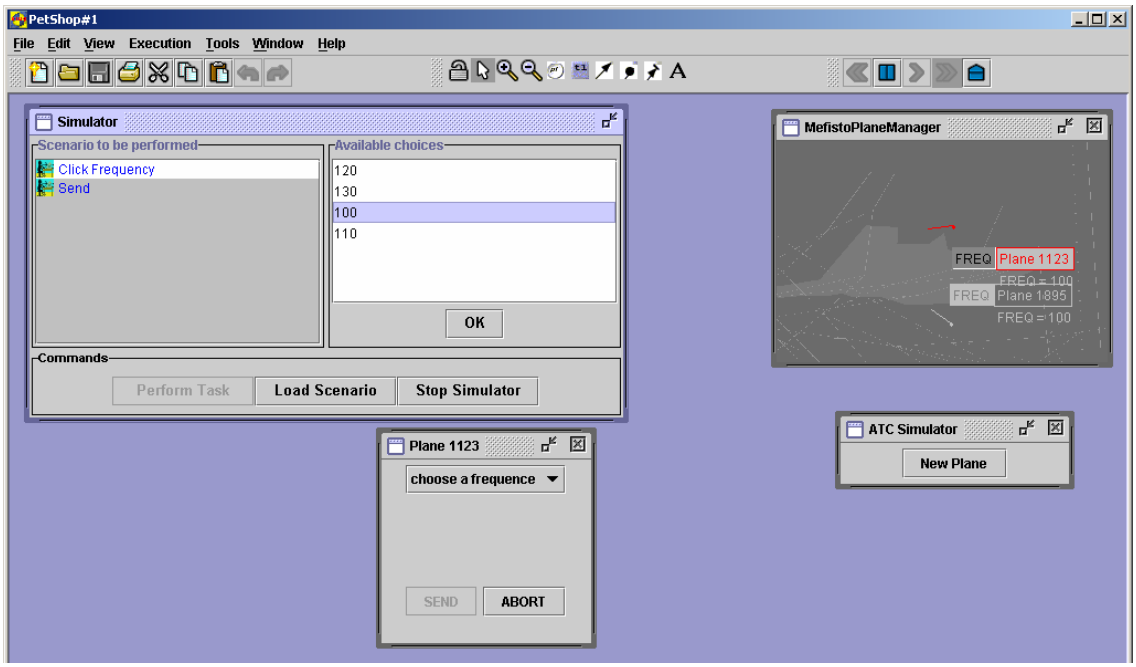

Figure 68. Interaction between Scenario and Current Execution: Values for Frequency are Selected at Runtime

Figure 68 shows the same interaction occurring while selecting a value for the frequency. The set of frequencies in the place Frequencies (see Figure 63) is displayed for user's selection in the scenario player.

The tool shows a dialogue window when the scenario has been successfully played on the description of the application using the ICO formalism. A scenario fails when at some point no action can be performed and the list of actions still to be performed is not empty.

9.3.4. Deviation-Based Evaluation

Evaluating a Basic Task (Current En-route, EXEC)

In this section we will focus on the executive controller task of transferring an aircraft, which, in a VHF-only environment, results in just managing the so-called 'last contact' with the pilot (namely, send the new frequency to the pilot). As we mentioned in Chapter 8, the analysis of deviations can be represented by tables structured as that below, which give the analyst or designer a means of recording the justification or rationale for decisions that have been made. In Table 13 we consider the task of sending the "last contact", instructing a pilot leaving the sector to contact the executive of the next sector. Each cause of deviation can be associated with the phase of the interaction cycle (intention, action, perception, understanding) when it occurs.

If we consider the "*none*" deviation (see Table 13) it can indicate either that the controller has not sent the clearance or that the clearance was sent by the controller but not received by the pilot. If both situations there might be a number of possible explanations (see "Causes" column in the Table), either on the controller's side or on the pilot's side.

Task: Handle VHF last contact		Guideword: None		
Deviation	**Causes**	**Consequences**	**Protections**	**Recommendations**
No message is sent by the controller	Controller fails to recognise the need for giving a new frequency (e.g. mistake due to failing to note aircraft location in proximity to sector boundary) Perception problem	No new frequency received by pilot — aircraft may enter the sector without having contacted the new controller	(i) If aircraft enters next sector without making a call, the next controller will call the current one. (ii) Pilot calls ATC	Provide an indication when aircraft within threshold distance of sector boundary Such an indication would persist until electronic system updated
	Controller recognises need, but fails to send message (e.g. memory lapse or high workload-induced slip) Action problem			
No message is received	Pilot fails to perceive message (e.g. inattention, high workload) Perception problem		(i) Controller expects answer; lack of it may prompt a second call. (ii) Pilot calls ATC	If there is no answer within some time limit an alarm message (e.g. an audible signal) could be automatically activated
	Total failure of communication technology			

Table 13: Example of Analysis of Deviations based on Guidewords

An "*other than*" deviation with the voice communication can be interpreted as meaning that, for some reason, the executive either sends the wrong information (wrong clearance, wrong parameter or both), or sends right information but it is incorrectly perceived by the pilot, or the wrong pilot is contacted. Wrong information can be given for various reasons. For example, a relevant environmental factor may have not been taken into account (e.g. other aircraft currently flying in the vicinity). Also, the *type* of solution may be correct but some of its details may not be. For example, the flight level chosen may not be appropriate for the direction of travel. Similar problems in a VHF-only environment can happen also because there is a lack of automatic tool support for making decisions. Thus all the support available for controllers, paper strips and radar information, require considerable cognitive effort. The pilot may misperceive or misinterpret the controller's command for various reasons, both linguistic and contextual.

Ill-timed communication is one that occurs either too early or too late. If the executive communicates a solution too early s/he can generate new problems other than solving the current one. For example, changing the flight level too early can generate a conflict with a flight that originally was not in conflict with that under consideration. Various reasons exist for a communication being late, most obviously, either controller may be busy with other duties. Another possibility is that when the executive needs to communicate, the radio channel is already "occupied" by another pilot. Indeed, another problem can be derived from limitations of radio communication. Since only one speaker can broadcast over the frequency at a time, the resulting communication has an asymmetrical nature (*one* speaker, *many* listeners). As all the pilots compete for the use of this resource, such sharing can be seen as penalising pilots.

Evaluating a Basic Task (Envisioned Aerodrome, GND)

In the prototype for the new environment that we evaluated it was already possible for the controller to zoom in or out the aerodrome map in order to have a more detailed view of some parts of the aerodrome (see Figure 69).

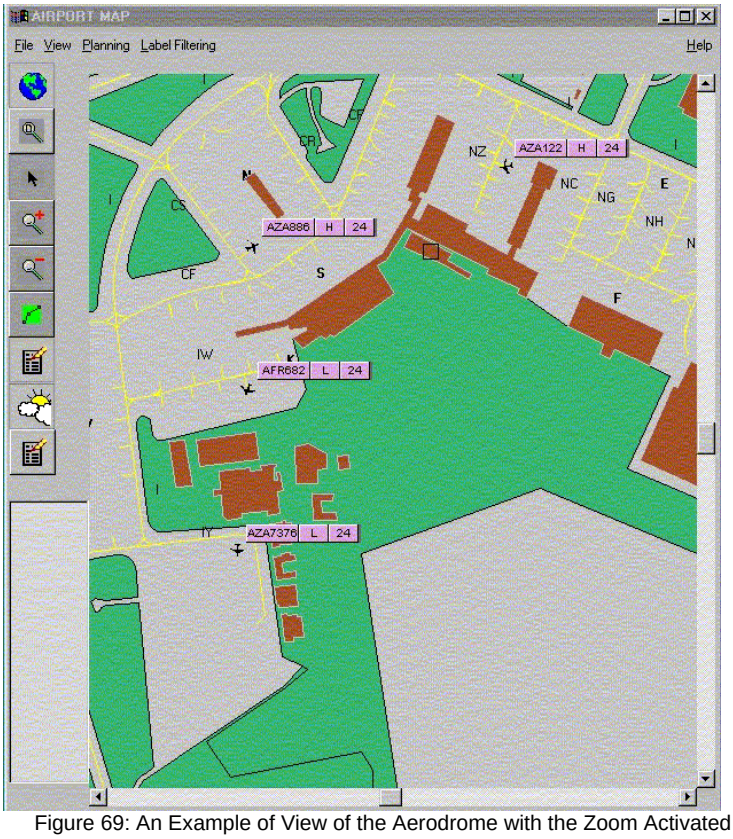

Figure 69: An Example of View of the Aerodrome with the Zoom Activated

However, it is possible that some conflicts occur in the part of the airport currently out of the controller's view (*No input* case of the *Check Deviation* task) because the controller has just zoomed in a different part of the aerodrome map. In this situation, the possibility of having a button to return to the default view is not sufficient to make the controller aware of some hazardous situations in time. Thus, the suggestion was to have permanently displayed the global view of the aerodrome map (for evident safety goals) and, on request, to activate a separate window highlighting a specific part of the aerodrome. For example in Figure 70 you can see a possible implementation of our suggestion where there is a secondary window presented after the controller requested to zoom in on a part of the aerodrome which is rather crowded of aircraft.

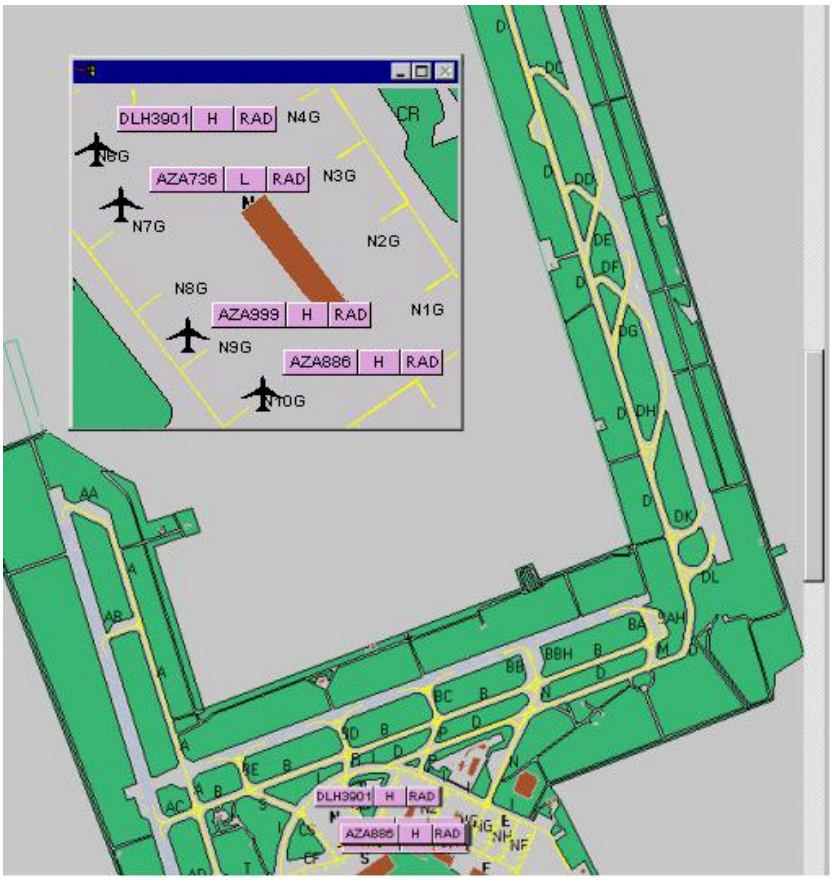

Figure 70. Possible Solution to the Zoom-In Problem

Evaluating a High Level Task (Envisioned Aerodrome, GND)

The interpretation and the application of each guideword to a higher-level task has to be properly customised depending on the temporal relationships existing between its subtasks. for example, if we consider the *Build path with automatic suggestions* task in Figure 71, because of the temporal relationship relating the first left subtask to the others, a lack of input for the first basic task means a lack of input for the whole parent task. Therefore, this case (*None/No input*) of the analysis of the parent task can be brought back to the correspondent case of its first left child. In this case it is the *SelectAutomaticModality* task that can be not performed because, for example, the controller does not see the related button. The *None/no performance* case means that no lower-level subtask (*Select Automatic Modality*, *Show suggested solutions* and *Select solution*) has been performed. The *None/No output* case is when all the subtasks have been carried out, but no output has been produced at the end. Referring to the aforementioned *Build path with automatic suggestions* task in Figure 71, this case can be brought back to the *None/no output* case of the *Select solution* task.

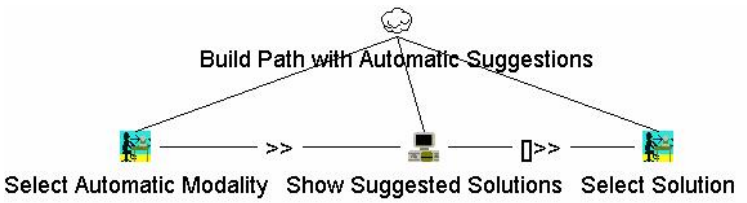

Figure 71. An Example of High-Level Task

For the other guidewords, the reasoning changes accordingly. For example, the *Less* guideword applied to the *Build path with automatic suggestions* task means that one associated sub-task has not been carried out. For example, the user forgets to perform the last subtask (*Select solution*) after the performance of the first two subtasks (*Select Automatic Modality* and

Show suggested solutions). The *More* case might arise because of an additional, unforeseen performance of one subtask (e.g., one task is executed more than once). The *Different* case occurs when a different temporal relationship is introduced within the subtasks (e.g., the subtasks are performed concurrently rather than sequentially) or wrong subtasks are performed. With regard to Figure 71, an example of a different temporal relationship would be for instance if the controller tries to select a solution before the system finishes displaying all of them (thus, *Show Suggested Solution* and *Select Solution* become concurrent tasks instead of sequential ones). The analysis of *Early* and *Late* guidewords should consider the parent task as a whole, inquiring about the impact on the system of all the situations when a too early/late performance of the parent task occurs.

10. Conclusions and Future Perspectives

The use of models has often been criticised for the effort required to develop them and the difficulties in using the information they contain to support design and evaluation. In order to alleviate this problem, a valuable solution is providing users with tools aiding the construction and analysis of such models.

Another traditional criticism against the use of models is that the benefits gained using them can rarely justify the cost of extra-time required. Again, this issue might be questionable as well because, even if the collection and analysis of such data could be time consuming, the costs of not performing such analyses seem even to increase, due to the higher probability of discovering and fixing errors later on in systems development.

In fact, especially in safety-critical applications the costs of formal specification are offset by the gains in safety consequent to detecting and preventing possible errors and defects in the design as early as possible, before actually using the system. Moreover, if we consider that such defects in safety-critical systems can even result in loss of human life, the overall cost-benefits ratio of using models seems to quickly become fairly acceptable.

In this dissertation we have shown a task model–based approach for developing interactive safety-critical systems, where task models succeeded in supporting different phases of the software development life cycle, from the design, to verification, to evaluation. Different techniques have been used in the various phases, all of them taking benefits from the expressiveness and flexibility provided by CTT notation and related CTTE tool.

The approach proposed was carried out within a project by researchers coming form the field of computer science, human factors and human-computer interaction. Domain experience was brought in by the air traffic control partners but of course other safety critical domains should be considered to generalise its applicability.

Also, still in the area of safety-critical interactive applications, and due to recent developments in wireless communication, increases in the power and interactive capabilities of portable devices, future work should be dedicated to further apply the method presented in this thesis to the design of interactive safety-critical systems exploiting the use of mobile devices in order to investigate whether, and in what way, new mobile technology could offer meaningful support to accomplish critical tasks while preserving safety and usability. In our group, some preliminary work has already started in this direction (Marhan *et al.*, 2004), by extending the analysis of deviations presented here and providing more explicit consideration of the distributed cognitive resources supporting task accomplishment.

References

(Abowd *et al.*, 1995)
Abowd, G., Wang, H., Monk, A., *A Formal Technique for Automated Dialogue Development*. Proceedings of Conference on Designing Interactive Systems (DIS 1995), ACM Press, pp.219-226.

(Annett and Duncan, 1967)
Annett, J., Duncan, K., *Task Analysis and Training in Design*. Occupational Psychology, 41, pp.211-221.

(Barclay *et al*., 1999)
Barclay, P., Griffiths, T., McKirfy, J., Paton, N., Cooper, R., and Kennedy, J., *The Teallach Tool: Using Models for Flexible User Interface Design*. Proceedings of International Conference on Computer-Aided Design of User Interfaces (CADUI 1999), pp. 139-158, 1999.

(Bastide and Palanque, 1990)
Bastide, R., and Palanque, P., *Petri Net Objects for the Design, Validation and Prototyping of User-Driver, Interfaces*. 3rd IFIP Conference on Human-Computer Interaction (INTERACT 1990), Cambridge, UK, August 1990, pp. 625-31.

(Bastide and Palanque, 1999)
Bastide, R., and Palanque, P., *A Visual and Formal Glue between Application and Interaction*. International Journal of Visual Language and Computing, vol. 10, no. 6, pp. 481-507, 1999.

(Baumeister *et al.*, 2000)
Baumeister, L., John, B., and Byrne, M., *A Comparison of Tools for Building GOMS Models*. Proceedings of Computer-Human Interaction Conference (CHI 2000), pp. 502-509, 2000.

(Beard *et al.*, 1996)
Beard, D., Smith, D. & Denelsbeck, K., *QGOMS: A Direct-Manipulation Tool for Simple GOMS Models*. Proceedings of Computer-Human Interaction Conference (CHI 1996), ACM Press, pp. 25–26, 1996.

(Berti and Paternò, 2003)
Berti, S., Paternò, F., *Model-based Design of Speech Interfaces*, Proceedings of Workshop on Design Specification and Verification of Interactive System (DSV-IS 2003). Springer Verlag, LNCS 2844, pp.231-244.

(Biere *et al.*, 1999)
Biere, M., Bomsdorf, B., Szwillus G., *Specification and Simulation of Task Models with VTMB*, Proceedings of Computer-Human Interaction Conference (CHI 1999), Extended Abstracts, pp.1-2.

(Bodart *et al.*, 1994)
Bodart F., Hennebert, A.-M., Leheureux, J-M., Provot, I., and Vanderdonckt, J., *A Model-Based Approach to Presentation: A Continuum From Task Analysis to Prototype*, Proceedings of 1st EUROGRAPHICS Workshop on Design, Specification and Verification of Interactive System (DSV-IS 1994), Bocca di Magra, Italy, 8-10 June 1994.

(Bolognesi and Brinksma, 1987)
Bolognesi, T., and Brinksma, E., *Introduction to the ISO Specification Language LOTOS*. Computer Network ISDN Systems, vol. 14, n. 1, pp. 25-59, 1987.

(Bomsdorf and Szwillus, 1999)
Bomsdorf B., Szwillus G., *Tool Support for Task-Based User Interface Design*, Proceedings of Computer-Human Interaction Conference (CHI 1999), Extended Abstracts, pp.169-170.

(Booch, 1994)
Booch, G., Object-Oriented Analysis and Design with Applications. Benjamin/Cummings, 1994.

(Booch *et al.*, 1999)
Booch, G., Rumbaugh, J., and Jacobson, I., Unified Modelling Language Reference Manual. Addison-Wesley, 1999.

(Burns and Pitblado, 1993)
Burns, D.J. and Pitblado, R.M., A *Modified HAZOP Methodology For Safety Critical Systems Assessment*. Directions in Proceedings of the Safety-Critical Systems Symposium, Bristol, 1993, Springer-Verlag.

(Cairns *et al.*, 2001)
Cairns, P., Jones, M., and Thimbleby, H., *Reusable Usability Analysis with Markov Models.* ACM Transactions on Computer Human Interaction (TOCHI), 8(2):99-132, 2001.

(Campos and Harrison, 2000)
Campos, J.C., Harrison, M., *The Role of Verification in Interactive Systems Design*. Proceedings of 5th International Workshop on Design, Specification and Verification of Interactive Systems (DSV-IS 2000), pp. 169-190, June 1998, Abingdon, UK, Springer Verlag.

(Campos and Harrison, 2001)
Campos J., Harrison D.M., *Model Checking Interactor Specifications*. Automated Software Engineering, 8, pp.275-310, 2001.

(Card *et al*., 1983)
Card, S., Moran, T., and Newell, A., The Psychology of Human-Computer Interaction. Hillsdale: Lawrence Erlbaum, 1983.

(Carroll 1995)
Carroll, J. (ed.), Scenario-based Design. John Wiley and Sons, 1995.

(Clarke *et al*., 1986)
Clarke, E.M., Emerson, E., and Sistla, A.P., *Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications*, ACM Transactions on Programming Languages and Systems, April 1986. 8(2), pp. 244-263.

(Curzon and Blandford, 2001)
Curzon P., and Blandford, A., *Detecting Multiple Classes of User Errors*. Proceedings of the 8th IFIP Working Conference on Engineering for Human-Computer Interaction (EHCI 2001), Lecture Notes in Computer Science, pp.57-72, Toronto, May 2001, Springer-Verlag.

(De Nicola *et al*., 1993)
De Nicola, R., Fantechi, A., Gnesi, S., and Ristori, G., *An Action-Based Framework for Verifying Logical and Behavioural Properties of Concurrent Systems*. Computer Network and ISDN Systems, 25, 1993, 761-77.

(Del Bimbo and Vicario, 1999)
Del Bimbo, A. Vicario, E., *A Visual Formalism for Computational Tree Logic*. Journal of Visual Language and Computing, v. 10, pp. 165-187, Academic Press, 1999.

(Dix, 1991)
Dix, A., Formal Methods for Interactive Systems. Academic Press, 1991.

(Dix *et al.*, 1998)
Dix, A. J., Finlay, J., Abowd, G.L., and Beale, R. Human-Computer Interaction. 2nd Ed., Prentice Hall, 1998.

(Duke and Harrison, 1993)
Duke, D., Harrison, M., *Towards a Theory of Interactors*. The Amodeus Project, Esprit Basic Research 7040, Amodeus Project Document, System Modelling/WP6, 1993.

(Duke and Harrison, 1994)
Duke, D., Harrison, M., *Folding Human Factors into Rigorous Development*. Proceedings of Eurographics Workshop on Design, Specification and Verification of Interactive Systems, F. Paternò (ed.), 335-352, 1994.

(Emerson and Lei, 1986)
Emerson, E. A., Lei, C. L., *Efficient Model-Checking in Fragments of the Propositional Mu-Calculus*. Proceedings of the 1st LICS, 267-278, 1986.

(Fernandez *et al*., 1996)
Fernandez, J., Garavel, H., Kerbrat, A., Mateescu, R., Mounier, L., and Sighireanu, M., *CADP: a Protocol Validation and Verification Toolbox*. Proceedings of the 8th Conference on Computer-Aided Verification (CAV 1996), pp. 437-440, 1996.

(Fields *et al*., 1997)
Fields, R.E., Harrison, M.D. and Wright, P.C., *THEA: Human Error Analysis for Requirements Definition*. University of York, Dept. of Computer Science, Technical Report YCS-97-294, 1997. http://www.cs.york.ac.uk/~bob/papers.html

(Foley and Sukavirya, 1994)

Foley, J. and Sukaviriya, N., History, *Results, and Bibliography of the User Interface Design Environment (UIDE), an Early Model-Based System for User Interface Design and Development*. Interactive Systems: Design, Specification, Verification, F. Paternò (ed.), pp. 3-14, 1994.

(Galliers *et al.*, 1999)
Galliers, J., Sutcliffe, A., Minocha, S., *An Impact Analysis Method for Safety-Critical User Interface Design*, ACM Transactions on Computer-Human Interaction, Vol.6, N.4, 1999.

(Garavel *et al.*, 2001)
Garavel, H., Lang, F., Mateescu, R., *An Overview of CADP 2001*. INRIA Technical Report TR-254, December 2001.

(Hartson and Gray, 1992)
Hartson, R., and Gray, P., *Temporal Aspects of Tasks in the User Action Notation*. Human-Computer Interaction, vol. 7, pp. 1-45, 1992.

(Hollnagel, 1993)
Hollnagel, R., Human Reliability Analysis —Context and Control. Academic Press, 1993.

(Ivory and Hearst, 1999)
Ivory, M., Hearst, M., *Comparing Performance and Usability Evaluation: New Methods for Automated Usability Assessment*, 1999. Report available at http://www.cs.berkeley.edu/~ivory/ research/web/papers/pe-ue.pdf

(Jacobson, 1992)
Jacobson, I. Object-Oriented Software Engineering. 2nd Ed. Addison-Wesley, 1992.

(John and Kieras, 1996)
John, B. E., Kieras, D. E., *The GOMS Family of User Interface Analysis Techniques: Comparison and Contrast*, ACM Transactions on Computer-Human Interaction (3), pp. 320–351, 1996.

(Johnson and Botting, 1999)
Johnson, C., and Botting, R., *Reason's Model of Organisational Accidents in Formalising Accident Reports*. Cognition, Technology and Work, Vol.1, N.3, pp.107-118, 1999.

(Lecerof and Paternò, 1998)
Lecerof, A., Paternò, F., *Automatic Support for Usability Evaluation*. IEEE Transactions on Software Engineering, Vol. 24, N.10, October 1998, IEEE Press, pp. 863-888.

(Leveson, 1995)
Leveson, N.G., Safeware: System Safety and Computers —A Guide to Preventing Accidents and Losses Caused by Technology. Addison Wesley, 1995.

(Leveson, 2000)
Leveson, N., *Intent Specifications: An Approach to Building Human-Centred Specification*. IEEE Transactions on Software Engineering, vol. 28, no. 1, pp. 15-35, Jan. 2000.

(Loer and Harrison, 2000)
Loer, K., Harrison, M., *Formal Interactive Systems Analysis and Usability Inspection Methods: Two Incompatible Worlds?*. Proceedings of the 7th International Workshop on Design, Specification and Verification of Interactive Systems (DSV-IS 2000). P. Palanque and F. Paternò (eds.), pp. 169-190, June 2000, Limerick, Springer Verlag.

(Lusini and Vicario, 1998)
Lusini, M., Vicario, E., *Engineering the Usability of Visual Formalisms: a Case Study in Real time Logics*. Proceedings of Advanced Visual Interfaces (AVI 1998), pp. 114-123, May 1998, L'Aquila, ACM Press.

(Marhan *et al.*, 2004)
Marhan, A.-M., Paternò, F., Santoro, C., *Designing Distributed Task Performance in Safety-Critical Systems Equipped With Mobile Devices*. Accepted for publication at Human Error, Safety and Systems Development Conference (HESSD 2004), Toulouse, August 22-27, 2004.

(Mateescu, 1998)
Mateescu, R., *Vérification des Proprietes Temporelles des Programmes Parallèles*. PhD Thesis, Institute National Polytechnique de Grenoble, April 1998.

(McDermid and Pumfrey, 1994)
McDermid J.A and Pumfrey D.J., *A Development of Hazard Analysis to aid Software Design*. Proceedings of COMPASS 1994, IEEE Press. ftp://ftp.cs.york.ac.uk/hise_reports/safety/develop.ps.Z

(MOD 1996)
UK Ministry of Defence (MOD), *HAZOP Studies on Systems Containing Programmable Electronics*. UK Ministry of Defence Interim Def Stan 00-58, 1996, Issue 1. Available at http://www.dstan.mod.uk/dstan_data/ix-00.htm

(Mori *et al.*, 2002)
Mori, G., Paternò, F., Santoro, C., *CTTE: Support for Developing and Analysing Task Models for Interactive System Design*. IEEE Transactions on Software Engineering, Vol. 28, No. 8, pp. 797-813, August 2002.

(Mori *et al.*, 2003)
Mori, G., Paternò, F., Santoro, C., *Tool Support for Designing Nomadic Applications*. Proceedings of ACM Intelligent User Interfaces (IUI 2003), Miami, pp.141-148, ACM Press, 2003.

(Mori *et al.*, 2004)
Mori, G., Paternò, F., Santoro, C., *Design and Development of Multi-Device User Interfaces through Multiple Logical Descriptions*. Accepted for publication on IEEE Transactions on Software Engineering, IEEE Press, 2004.

(Mullet and Sano, 1995)
Mullet, K., Sano, D., Designing Visual Interfaces. Prentice Hall, 1995

(Navarre *et al.*, 2000)
Navarre, D., Palanque, P., Bastide, R., Sy, O., *Specification of Middles Touch Screen using Interactive Cooperative Objects*. MEFISTO Project Working Paper 2.5, 2000 http://giove.cnuce.cnr.it/mefisto/wp2_5.html

(Newman and Lamming, 1995)
Newman, W. and Lamming, M. Interactive System Design. ISBN 0-201-63162-8, Addison Wesley, 1995.

(Nielsen, 1993)
Nielsen, J., Usability Engineering. Boston, Academic Press, 1993.

(Norman, 1988)
Norman, D., The Psychology of Everyday Things. Basic Books, 1988.

(Object Management Group, 1998)
Object Management Group, *The Common Object Request Broker: Architecture and Specification*. CORBA IIOP 2.2 /98-02-01, Framingham, MA, 1998.

(Palanque and Bastide, 1990)
Palanque, P., Bastide, R., *Petri Net with Objects for the Design, Validation and Prototyping of User-Driven Interfaces*. Proceedings of the 3rd IFIP Conference on Human-Computer Interaction (INTERACT 1990), D. Diaper, D. Gilmore, G. Cockton and B. Shackel (eds.), pp. 625-63, Cambridge, Elsevier Science.

(Palanque and Bastide, 1994)
Palanque, P., Bastide, R., *Petri Net –Based Design of User-Driven Interfaces Using the Interactive Cooperative Objects Formalism*. Interactive Systems: Design, Specification, Verification. Springer Verlag, pp. 383-400, 1994.

(Palanque and Bastide, 1997)
Palanque, P., and Bastide, R., *Synergistic Modelling of Tasks, Users and Systems Using Formal Specification Techniques*. Interacting With Computers 9, N. 2, pp. 129-53, 1997.

(Palanque *et al.*, 1995)
Palanque, P., Bastide, R., and Sengès, V., *Validating Interactive System Design Through the Verification of Formal Task and System Models*. 6th IFIP Conference on Engineering for Human-Computer Interaction, (EHCI 1995), Garn Targhee Resort, Wyoming, USA, August 14-18. Chapman et Hall, 1995.

(Palanque *et al.*, 1997)
Palanque, P., Bastide, R., and Paternò, F., *Formal Specification As a Tool for the Objective Assessment of Safety Critical Interactive Systems*. Proceedings of the 6th IFIP TC13 Conference on Human-Computer Interaction (INTERACT 1997), Sydney, Australia, 14-18 July 1997, 323-30. Chapman & Hall.

(Paternò, 1994)
Paternò, F., *A Theory of User-Interaction Objects*. Journal of Visual Languages and Computing, Academic Press, Vol.5, N.3., pp.227-249.

(Paternò, 1999)

Paternò, F., Model-Based Design and Evaluation of Interactive Application. Springer Verlag, ISBN 1-85233-155-0, 1999.

(Paternò, 2000)
Paternò F., *The Specification of the ConcurTaskTrees Notation*. GUITARE Working Paper, March 2000.

(Paternò, 2001)
Paternò, F., *Towards a UML for Interactive Systems*. Proceedings of Engineering for Human-Computer Interaction Conference (EHCI 2001), pp. 175-185, 2001.

(Paternò *et al.*, 1998)
Paternò, F., Breedvelt-Schouten, I., de Konig, N., *Deriving Presentations from Task Models.* Proceedings of Engineering for Human-Computer Interaction (EHCI'98), Creete, Kluiwer Publisher, 1998.

(Paternò *et al.*, 2000)
Paternò, F., Santoro, C., Sabbatino, V., *Using Information in Task Models to Support Design of Interactive Safety-Critical Applications*. Proceedings of Advanced Visual Interfaces (AVI 2000), pp.120-127, May 2000, Palermo, Italy, ACM Press.

(Paternò and Leonardi, 1994)
Paternò, F., Leonardi, A., *A Semantics-based Approach to the Design and Implementation of Interaction Objects*. Computer Graphics Forum, Blackwell Publisher, Vol.13, N.3, pp.195-204, 1994.

(Paternò and Mezzanotte, 1995)
Paternò, F., Mezzanotte, M., *Formal Verification of Undesired Behaviours in the CERD Case Study*. Proceedings of Engineering of Human Computer Interaction (EHCI 1995), L. Bass and C. Unger (eds.), pp.213-226, Wyoming, Chapman & Hall, August 1995.

(Paternò and Santoro, 2001)
Paternò, F., Santoro C., *User Interface Evaluation When User Errors May Have Safety-Critical Effects*. Proceedings of the 8th IFIP TC13 Conference on Human-Computer Interaction (INTERACT 2001), Tokio, Japan.

(Polson et al., 1992)
Polson, P., Lewis, C., Rieman, J., & Wharton, C. (1992). *Cognitive walkthroughs: A method for theory-based evaluation of user interfaces.* International Journal of Man-Machine Studies, 36, 5, 741-773.

(Preece *et al.*, 1994)
Preece, J., Rogers, Y., Sharp, H., Benyon, D., Holland, S., and Carey, T. Human-Computer Interaction. Cambridge University Press, Addison Wesley, 1994.

(Puerta, 1997)
Puerta, A., A Model-Based Interface Development Environment. *IEEE Software*, pp. 40-47, July/August 1997.

(Puerta and Eisenstein, 1999)
Puerta, A.R. and Eisenstein, J., *Towards a General Computational Framework for Model-Based Interface Development Systems*. Proceedings of International Conference on Intelligent User Interfaces (IUI 1999), pp.171-178, ACM Press, January 1999.

(Rational, 1997)
Rational Software Corporation. UML Notation Guide. 1.1, September 1997.

(Reason, 1990)
Reason, J., Human Error. Cambridge University Press, 1990.

(Rettig, 1994)
Rettig, M. *Prototyping for Tiny Fingers*. Communication of the ACM. 37(4), pp. 21-27, April 1994.

(Rumbaugh et al., 1997)
Rumbaugh, J., Jacobson, I., and Booch, G., Unified Modeling Language Reference Manual, ISBN: 0-201-30998-X, Addison Wesley, December 1997.

(Rushby, 1999)
Rushby, J., *Using Model Checking to Help Discover Mode Confusion and Other Automation Surprises*. Proceedings of the 3rd Workshop on Human Error, Safety, and System Development (HESSD 1999), University of Liege, Belgium, June 1999.

(Scapin and Pierret-Golbreich, 1989)

Scapin, D., and Pierret-Golbreich, C., *Towards a Method for Task Description: MAD*. Work with Display Units, (89), pp. 371-380, 1989.

(Sutcliffe, 1997)
Sutcliffe, A., *Task-Related Information Analysis*. International Journal on Human-Computer Studies, vol. 47, pp. 223-257, 1997.

(Sy *et al.*, 1999)
Sy, O., Bastide, R., Palanque, P., Le, D.-H., and Navarre, D., *PetShop: a CASE Tool for the Petri Net Based Specification and Prototyping of CORBA Systems*. Proceedings of 20th International Conference on Applications and Theory of Petri Nets (ICATPN 1999), Williamsburg, VA, USA, 1999.

(Szekely *et al.*, 1993)
Szekely, P., Luo, P., and Neches, R., *Beyond Interface Builders: Model-Based Interface Tools*. Proceedings of Conference Human Factors and Computing Systems (INTERCHI 1993), pp. 383-390, 1993.

(Tauber, 1990)
Tauber, M. (1990), *ETAG: Extended Task Action Grammar - A Language for the Description of the User's Task Language*, D. Diaper, D. Gilmore, G. Cockton & B. Shackel, eds, Proceedings of INTERACT '90, Elsevier, Amsterdam, 1990.

(Tardieu *et al.*, 1983)
Tardieu, H., Rochfeld, S., and Coletti, R. *La Méthode MERISE*. Organisations, 1983.

(van der Veer *et al.*, 1996)
van der Veer, G., Lenting, B., and Bergevoet, B., *GTA: Groupware Task Analysis and Modelling Complexity*. Acta Psychologica, vol. 91, pp. 297-322, 1996.

(van Welie, 2001)
van Welie, M., *Task-Based User Interface Design*. PhD Thesis, SIKS, 2001.

(van Welie *et al.*, 1998a)
van Welie, M., van der Veer, G.C., and EliëÈns, A., *An Ontology for Task World Models*. Proceedings of the 5th International Eurographics Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS 1998), pp. 57-70, 1998.

(van Welie *et al.*, 1998b)
van Welie, M., van der Veer, G., and EliëÈns, A., *Euterpe - Tool Support for Analyzing Cooperative Environments*. Proceedings of the 9th European Conference on Cognitive Ergonomics, Limerick, Ireland, pp. 25–30, 1998.

(Wharton *et al.*, 1994)
Wharton, C., Rieman, J., Lewis, C., and Polson, P., *The Cognitive Walkthrough: A Practitioner's Guide*. In Usability Inspection Methods, J. Nielsen and R.L. Mack (eds.), John Wiley & Sons, 1994.

(Wilson *et al.*, 1993)
Wilson, S., Johnson, P., Kelly, C., Cunningham, J., and Markopoulos, P., *Beyond Hacking: A Model-Based Approach to User Interface Design*. Proceedings of Human-Computer Interaction Conference (HCI 1993), pp. 40-48, 1993.

(W3C, 2003)
http://www.w3.org/MarkUp/Forms/. W3C Recommendation 14 October 2003.

List of Figures

List of Tables