Support for Reasoning about Interactive Systems through HCI Designers' Representations

Fabio Paternò, Carmen Santoro

ISTI-CNR, Via G.Moruzzi 1
56100 Pisa, Italy
{F.Paterno, C.Santoro}@cnuce.cnr.it

Abstract. In this paper we present and discuss the integration of tools for the development and analysis of human-computer interaction models with formal verification techniques. We explain the method that we have developed and the tools that we have integrated for this purpose, and discuss how the resulting environment is more effective for designers of interactive applications than using current formal verification techniques alone. The main result is that designers can take advantage of the functionality of model-checking tools while still working with representations of the relevant models that they are familiar with in their practice. Thus, we show how tools can support such an approach 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.

Keywords: Models for Human-Computer Interaction, Formal methods for human-computer interaction, Task and dialogue-related properties.

1. Introduction

The importance of the user interface and its related software in interactive software applications is increasingly being recognised. Usability is one fundamental aspect of most interactive applications [8]. For example, it is useless to implement very efficient functions if users have to spend a lot of time to understand how to access them (or may not even be able to access them at all).

In the area of human-computer interaction (HCI), approaches often lack systematic and engineered methods, with precise concepts. However, formal methods in this area have stimulated limited interest, particularly in industrial environments. The major concern is that dealing with them is usually too difficult, as even related tools seem to offer insufficient support to allow their easy and effective use.

In hardware design, where it is important to check that some properties are satisfied before implementing the specification into hardware, formal verification has been successfully and widely used. The HCI field is more challenging for verification methods and tools, since the specification of human-computer dialogues may be more complex than the hardware specifications. The reason for such complexity is the human behaviour that can be affected by many factors. In addition, since formal approaches require formally describing an interactive system and identifying significant properties, it is necessary 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. Thus, the problem of how to facilitate their acceptance is multi-faceted.

Early work in the area of formal methods for human-computer interaction mainly aimed to formally express relevant concepts [13]. Then, interest in a more systematic introduction of formal techniques in the design cycle has arisen. For example, the importance of formal specification for the design and implementation of interactive applications has already been pointed out. 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 [1, 5, 23, 27, 29]. A kind of model checking approach is also the work by Thimbleby and others [4] where Markov models are analysed through "knowledge/usability graphs" even if in this approach the demonstration of formal properties is not considered.

Indeed, there are various motivations to carry out model checking for user interfaces:

- 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 useful analysis.
- User testing can be rather expensive, particularly in highly specialised fields such as ATC (Air Traffic Control), the users are 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. We are not proposing that users should not be involved in testing, but we believe that model checking can decrease the number of problems found at the stage of empirical testing, thereby reducing expensive changes late in the design process, although empirical testing will always be useful.
- Model checking allows an exhaustive analysis; the advantage of model checking is that the space of the states reachable by the specification is completely analysed. 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. This lack of completeness in empirical testing can have dangerous effects especially in safety-critical contexts.

A few works have addressed the issue of supporting designers while formalising properties. For example, in [20] a visual formalism for the presentation of a real-time logic is proposed. It is based on a visual metaphor allowing mapping textual sentences onto visual representations. The resulting notation is supported by an interactive syntaxdriven editor integrating the visual presentation with the conventional textual notation. Also in [11] a visual system is described which supports visual presentation and manipulation of temporal Computation Tree Logic (CTL) [7] 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 prove difficult to manage. In order to ease their application, a number of new tools have been proposed. For example, the Preventing User Errors project [9] regards 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 [6], 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 [19], 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 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 people very often need a good knowledge of those notations to be able to specify properties. In addition, even when they know a given language, sometimes there are additional constraints that 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.

The goal of this paper is to present a method aiming to support the introduction of formal techniques in the design cycle of interactive systems and an environment supporting such 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. 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. To illustrate this approach we also consider an example in the area of air traffic control. In fact, it is in such safety-critical applications that the costs of formal verification are offset by the gains in safety consequent to detecting and preventing possible errors before actually using the system.

More specifically, in the paper we first outline the approach proposed, then we give a more structured description of the various phases composing the method. Next, we show how the integration of a tool for task modelling and a formal model checker has been achieved, also presenting and discussing an example of application taken from the case study considered. Lastly, some concluding remarks are drawn.

2. The approach

An interactive application is characterised 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. The introduction of a formal representation has to be motivated: the effort to formalise presentation aspects does not seem to be justified because we obtain models that describe features that can be easily understood by direct inspection of the implemented user interface, as they are strictly related to how people perceive information. The case of user interface dialogues, 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 many possible paths of actions. However, the separation of presentation from dialogue poses problems and there are cases where important user interface issues involve both lexical/presentation and dialogue issues [14] and our approach allows capturing such relationships. 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 is to increase even more such concurrency. On the one hand, it is a source of flexibility, usability, and interactive richness but, on the other hand, 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, a designer needs first to understand the logical and temporal relationships among the possible logical activities (which to some extent depend on the artefacts available) and, consequently, 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 paper 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.*

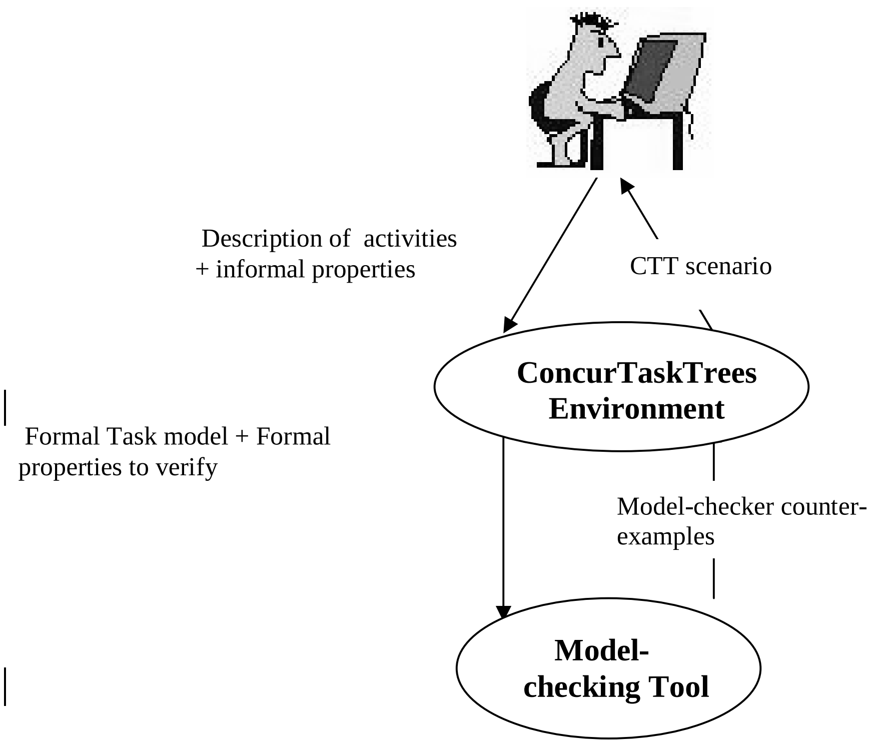

Description of activities
+ informal properties
CTT scenario
ConcurTaskTrees Environment
Formal Task model + Formal properties to verify
Model-checker counter-examples
Model-checking Tool

Figure 1: Input and Output flows between CTTE and Model-checker

To summarise, the main objective of our method (see Figure 1 above) 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 use because they are not able to reflect back problems in the proof in an intelligible form. By using an extension of the ConcurTaskTrees Environment (CTTE) [22], 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. 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.

The method that we propose for the design of interactive systems with the support of formal methods is represented in Figure 2, where the processes are indicated with ovals and the results with rectangles. The ovals filled by points are tool-supported and are the phases that we mainly discuss in this paper. We have developed most of the tools involved in the method: more specifically, the phases concerning task modelling, CTT-to-LOTOS transformation and property formalisation are included within the CTTE tool, whereas all the other tool-supported phases (model-checker transformation and property verification) are provided by a model-checker developed by other groups [16].

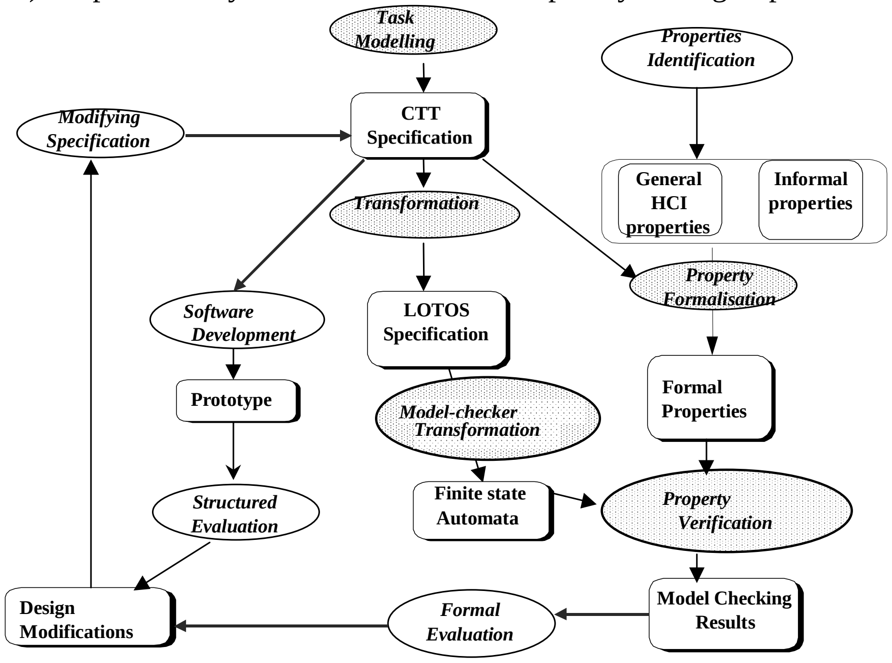


Figure 2: The method.

More precisely, the method comprises the following activities:

- *Task Modelling*. After a first phase gathering information on the application domain, and an informal task analysis, designers should develop a task model that forces them to clarify many aspects related to possible tasks and their relationships. The task specification is obtained by first structuring the tasks in a hierarchical way so that abstract tasks are described in terms of more refined tasks. Next, temporal relationships among tasks are described using ConcurTaskTrees (CTT) [24], which is a graphical notation allowing designers to describe hierarchies of tasks with temporal operators that have been identified extending the basic set of LOTOS operators. In this way, we obtain a ConcurTaskTrees 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 thus giving a clear indication of how cooperations are performed. For each task it is also possible to provide further information concerning the objects manipulated and attributes such as frequency, time requested and so on.
- *Software Development.* The task model can be used to drive the development or the analysis of a software prototype, in particular, presentation prototypes, consistent with the indicated requirements (for example, if we know that the application has to provide an overview of some information, then suitable presentation techniques for summarising the data should be considered). However, the problem of how to derive a concrete interface model from the abstract specification is out of the scope of this paper. This issue has been addressed in various works (see for example [28]).
- *ConcurTaskTrees-to-LOTOS Transformation*. The other possibility is transforming the ConcurTaskTrees model into a LOTOS specification to analyse potential task performance and the user interface dialogues by using model checking techniques. We have implemented a transformation tool where each task specification is translated into a corresponding LOTOS process. The reasons for this transformation is that there are various model-checking tools able to accept LOTOS specifications as input (such as the CADP package [16] we used in our work).
- *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 that are 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 [17] which translates the behavioural part of a LOTOS specification into a Labelled Transition System (LTS). The property verification phase is performed by the EVALUATOR model-checker [21], 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 thus restarting the process.

4. Representing Task Models with ConcurTaskTrees notation

One of the key aspects to obtaining usable systems is to design user interfaces that provide an immediate mapping between the actions and the representations they provide and the tasks that should be performed. Most usability problems arise when there is a mismatch between how users think that tasks should be performed (user task model) and how the system actually supports such performance.
Task models allow designers to describe the set of activities that can be performed in order to reach the user's goals. They can be useful in various phases of the design cycle but, above all, the process of specifying activities within the task modelling stage helps designers to clarify (also to themselves) the relationships among the different activities that should be performed in the envisioned system and in this way designers can acquire a better understanding of the different design choices. In addition, they allow the designer to specify and obtain an in-depth insight, with no ambiguity, of the considered interactive system's behaviour. However, in spite of such advantages, this process is often criticised because it is a time-consuming, sometimes discouraging, activity.
However, the use of tools can strongly decrease such costs as they also facilitate reuse of the specification across various phases, including documentation [31]. In addition, especially in specific domain areas, such costs can easily become more than acceptable if they are compensated by the gained benefits. For example, it is the case of interactive safety-critical systems, where the best trade-off is achieved between costs (in terms of people, time and effort involved in this process) and benefits (in terms of safety gained). In fact, in such systems where even a small failure might have unacceptable consequences, and where user actions sometimes cannot be undone (for example, if an irreversible physical process has been activated), it is important to guarantee that the user interface is able to support users while they carry out the expected tasks. Task models represent the logical and temporal relationships among tasks and allow evaluators to identify any unexpected user behaviour, which is very often an indication of problematic areas which usability and safety analyses should be concerned with. It is evident that in such systems the issue of user errors and how to design the user interface so as to avoid them acquires a special importance. The importance of understanding the reasons of human errors has been highlighted in many studies: in [18] a number of techniques useful to better understand the human errors that have caused real accidents have been developed; they have also shown that accidents often are caused by human errors whose likelihood may be increased by poor design.

The ConcurTaskTrees (CTT) notation [24, 25] was introduced after first experiences [26] in specifying graphical user interfaces with LOTOS, a concurrent formal notation that seemed at first a good solution to describe user interfaces, as it allows designers to describe both event-driven behaviours and state modifications. However, we soon realised the inadequacy of LOTOS within the HCI domain: for example, its textual notation can easily generate complex expressions even when the behaviour to be described is quite simple. In addition, we noticed that other notations for task models were lacking in precise semantics or constructs useful for obtaining flexible descriptions. Thus, we realised there was a need for a new notation with operators able to rigorously and succinctly express a richer set of dynamic behaviours in task models. This would allow designers to focus on aspects more relevant for user interface design and avoid getting lost in detailing complicated specifications that are difficult to understand and follow even for themselves. This notation, ConcurTaskTrees, aims to be an easy-to-use notation that can support the design of real industrial applications, which usually means medium- to large-sized applications. In CTT a task model is composed of a set of concurrent tasks, possibly consisting of several sub-tasks. A sub-task is a task in itself, so that in general a CTT specification describes a system via a hierarchy of task definitions, like in the Hierarchical Task Analysis (HTA) notation [2].

In CTT, different icons are used to indicate how the performance of a task is allocated, and each of those icons corresponds to a specific ‘category’. We have basically four categories:

- *Interaction,* tasks performed by the user interacting with the system;
- *Application,* tasks entirely executed by the application. For example, providing feedback about the current state of the system.
- *User*, cognitive activities performed by the user. Those tasks are used when designers want to highlight the performance of a cognitive task on the user’s side.
- *Abstraction,* tasks associated with complex activities whose performance cannot be univocally allocated.

In CTT the state representation is highlighted by means of application tasks. For example, showing something on the screen is an application task.
The leaves of the tree are the basic tasks that cannot be further decomposed. Sibling tasks are connected by means of operators. A task inherits the temporal relationships of the parent and ancestors tasks. The semantics of these operators provides the rules for the interactive system temporal evolution. Table 1 indicates the temporal operators used in ConcurTaskTrees. It is worth noting that the CTTE tool allows designers to break down higher level tasks into subtasks without having to specify immediately temporal operators.

Some work has been done [3] about the opportunity of extending the CTT notation in order to express a set of *task patterns,* which are task structures that incorporate design solutions to recurring problems. In fact, the hierarchical structure of ConcurTaskTrees facilitates this development as it provides a wide range of granularity allowing large and small task structures to be reused. In addition, the CTTE tool supports automatic expansion of task patterns. So, once a task has been defined with all the subtasks and designers want to include it in various points of the model, then it is sufficient to indicate the name of the task pattern where they want to include it and the tool automatically expands it in all the occurrences.

Independent Concurrency (T1 ||| T2)	Actions belonging to two tasks can be performed in any order without any specific constraints, for example monitoring a screen and speaking in a microphone;
Choice (T1 [] T2)	It is possible to choose from a set of tasks and, once the choice has been made the task chosen can be performed and other tasks are not available at least until it has been terminated. This operator is useful in the design of user interfaces because it is often important to enable the user to choose from various tasks. An example is, at the beginning of a word processor session when it is possible to choose whether to open an existing file or a new one. Also the system can choose to perform one task from a set of application tasks depending on its current state.
Concurrency with information exchange (T1|[]| T2)	Two tasks can be executed concurrently but they have to synchronise in order to exchange information. For example, in current air traffic control centres each controller has to manage pilots' requests and communicate with other controllers in the same center. Then, not only are these two activites concurrently performed but also information (about flights) is continuously exchanged between them.
Order Independence (T1 |=| T2)	Both tasks have to be performed but when one is started then it has to be finished before starting the second one.
Deactivation (T1 [> T2)	The first task is definitively deactivated once the first action of the second task has been performed. This concept is useful for modelling situations in which, for example, the user is in the middle of a dialogue box and then abandons it by selecting the 'cancel' button.
Enabling (T1 >> T2)	This operator, as well as the following one, models a sequential composition between two activities: when the first task terminates, it enables the performance of the second one. For example, a database application where users have first to register and then they can interact with the data.
Enabling with information passing (T1 []>>T2),	In this case task T1 provides some information to task T2 besides enabling it. For example, T1 allows the user to specify a query and T2 provides the query result that obviously depends on the information generated by T1.
Suspend-resume (T1 |> T2),	This operator gives T2 the possibility of interrupting T1 and then when T2 is terminated, T1 can be reactivated from the state reached before the interruption. For example, the editing text task which, in some applications can be suspended by a modal printing task, and once the printing task is accomplished then editing can be carried on from the state reached beforehand. For example, this operator can be used to model a type of interruption.
Iteration T*	In the tasks' specification we can have some tasks with the * symbol next to their name. This means that the tasks are performed repetitively: when they terminate, the performance of their actions automatically starts to be executed again from the beginning. This continues until the task is deactivated by another task.
Finite Iteration (T1(n))	It is used when designers know in advance how many times a task will be performed.
Optional tasks ([T])	They give the possibility of indicating that the performance of a task is optional. Optional tasks are indicated in square brackets. For example, we have optional tasks when we fill a form in and there are some fields that are mandatory and others optional.
Recursion	In the CTT notation, no specific operator is associated with recursion. A high level task T is recursive when an occurrence of it appears within its own specification. This possibility is used, for example, with tasks that, for each recursion, allows performance of the recursive tasks with the additional possibility of performing some new tasks, until a task interrupting the recursion is performed.

Table 1: Temporal operators in ConcurTaskTrees.

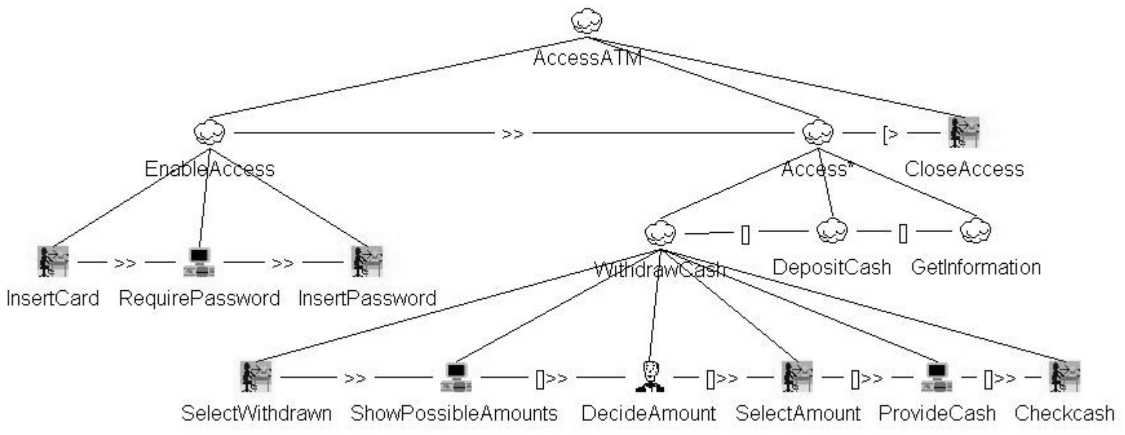


Figure 3: Example of CTT specification.

In Figure 3 an example of a task model describing the interactions of a user with an ATM machine is shown to illustrate the basic features of the notation. Users can access their accounts after inserting an ATM card and typing the password. Then, they have the possibility of choosing among a set of high level activities, e.g. deposit cash, get information about the account, withdraw cash. If users want to withdraw cash they have to decide (cognitive task in the task model, see the associated icon) and select (the interaction task 'SelectAmount' has been included in the task model) the appropriate amount of money so that the ATM machine can supply them with it. At any time, the user can decide to abort the transaction and leave the system (see the '[>' operator on the left of the "CloseAccess" task).

In addition, ConcurTaskTrees allows designers to specify multi-user applications. In these cases the task model is composed of various parts. There is one task model for each role involved (where a role is identified by a specific set of tasks and relationships among them) and, there is also a cooperative part describing the relationships among the 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*, which are 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, we can identify connection tasks because they are annotated with a double arrow under their names.

It is worth pointing out that, in order to enable a connection task it is necessary that not only the 'local' constraints be verified (namely, the constraints they have towards the tasks belonging to the same role), but also the constraints they have towards the tasks belonging to other roles. So, it can happen that a task sequence is unreachable because there are connection tasks that simply will never be enabled due to constraints they have towards tasks belonging to other roles.

5. From ConcurTaskTrees to LOTOS

In order to perform model checking of CTT task models we translate such models into LOTOS. A different approach for a similar problem was followed in [5]. In that case authors had Interactor behaviours described using a logic based on Structured MAL and they had to convert it into a SMV specification. Then, they used a model checker that allowed them to express CTL (Computational Tree Logic) properties of the behaviour of the system specified in SMV.

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 to a LOTOS process*. It is worth noting that in the translation of CTT task every information about task attributes (category, type, etc.) has been neglected when translating in LOTOS.

Consider the structure of a typical LOTOS specification:

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

Fig. 4: Typical structures of specification and process definitions in LOTOS.

Apart from *type definitions* (e.g. Boolean, Natural, ..) and *parameter list* (which is a list of variable declarations) which both are included in full LOTOS specifications for expressing data values, but which were not used in our case, 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 which have 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, and with Process_definition(P) the definition of process P.

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 has not to be explicitly defined.

5.1 Translation of 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.

<u>Rule 1) T leaf task, not iterative</u>

T (in CTT) *translates to* :

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 will be listed all the elementary actions appearing in its sub-processes.

<u>Rule 2) T high level task, T= t_1 op_1 t_2 … op_m t_{m+1} with t_1, t_2 … , t_{m+1} children of T</u>

T (in CTT) *translates to* :

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

```
            t1_proc [gates(t1)]  op1 …    opm  tm+1_proc [gates(of tm+1)]
        where
            < Process_definition(t1)>
             …
            < Process_definition(tm+1)>
 endproc
```

Once the ConcurTaskTrees tasks are mapped onto LOTOS processes we have to define how to express the ConcurTaskTrees temporal operators in terms of LOTOS expressions.

Regarding the translation of the ConcurTaskTrees operators, some of them derive directly from LOTOS (this is the case for 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:

Rule 3) *Optional* [T]: 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 by a choice operator.
Optional [T]
Behaviour expression of [T] =
T_proc[gates(T)] [] exit

Rule 4) *Iteration* T*: the implementation of an iterative 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* (in CTT) *translates to* :

process T_proc[T]:noexit:= T; T_proc[T] **endproc**

In case T is a high level task, T iterative, T= t_1 op_1 t_2 … op_m t_{m+1} with t_1, t_2 … , t_{m+1} subtasks of T
T* (in CTT) *translates to* :

```
        process T_proc[gates(T)]:noexit:=
            (t1_proc [gates(t1)]  op1 …    opm  tm+1_proc [gates(tm+1)])
            >>
            T_proc[gates(T)]
```

```
 where
        < Process_definition(t1)>
         …
        < Process_definition(tm+1)>
endproc
```

<u>Rule 5)</u> <u>*Order-independence* (|=|):</u> if two tasks A and B are connected with this operator, the overall 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.

```
behaviour_expression ([T1 |=| T2])=

(T1_proc[gates(T1)]  >> T2_proc[gates(T2)] )
[]
(T2_proc[gates(T2)] >> T1_proc[gates(T1)])
```

<u>Rule 6)</u> *Suspend-resume operator* (|>), 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 t_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 t_i [i= 1, …,n] of A with another expression which is the choice between t_i and B, with B followed by the recursive instantiation of such choice. This means that if t_i is performed, we can carry out the next basic task of A (t_{i+1}), otherwise B is performed and then it is still possible to choose one of the two possibilities. In algorithmic terms this means:

1. Identification of the basic tasks associated with A and the first basic task to be performed ;
2. In the corresponding LOTOS expression, where the action associated with the current basic task should occur, an expression is inserted composed of a choice between either the action associated with the current basic task or the LOTOS process associated with B followed by the recursive instantiation of such a choice;
3. If the final basic task of A has been considered, then stop, otherwise consider the next basic task of A to be performed and go on to 2.

5.2 Translation of Cooperative task models

In the previous section we have 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 this step further to complete this transformation if we consider cooperative CTT task trees. 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 interprocess 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 we have still to do 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 following specification:

specification Coop_spec [*gate list*] : *functionality*

behaviour
 behaviour expression (1)
where
 process definitions
endspec

where:

- *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).*

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 the following:

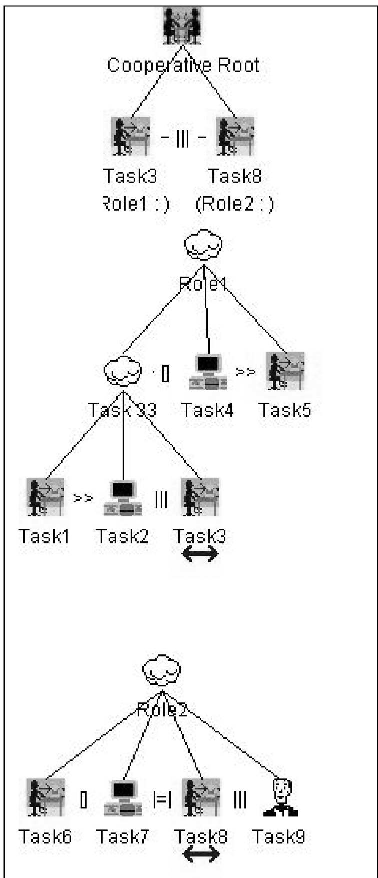
Cooperative_Root_proc[*gates(Coop)*]
 |[*gates(Coop)*]|
(
 Role_1_proc[*gates (Role_1)*]
 |||
…
 Role_n_proc[*gates (Role_n)*]
)

where *gates (Coop)* are the elementary actions of the Cooperative part (in other words, the CTT connection tasks), and for each i=1, …, n *gates (Role_i)* are the gates of each Role_i.

In order to better explain the rules driving this translation, an example of a cooperative task model with the corresponding LOTOS translation is shown in figure 5. As you can be seen, 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:

```
(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 5). 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 5).



```
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 5: Structure of the transformation from ConcurTaskTrees to LOTOS.

6. Property Formalisation and Verification in the Integrated Environment

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) [15] 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. 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 [12] operators that are mapped onto MCL constructs. Examples of properties considered are:

Permanent Reachability (task)
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 (task)
The goal of this template is to check whether in any state of the system exists a temporal evolution (E operator) during which it is possible to reach a state (F operator) where *task* can be executed. The corresponding formal template is 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 (task1, task2)
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(task1, task2)
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: 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).

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.

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 2: Examples of property templates identified

7. 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 (ConcurTaskTrees Environment) that supports editing, analysis and simulation of task models of cooperative applications.
- A model-checking tool: we used that included in the CADP package but 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 6).

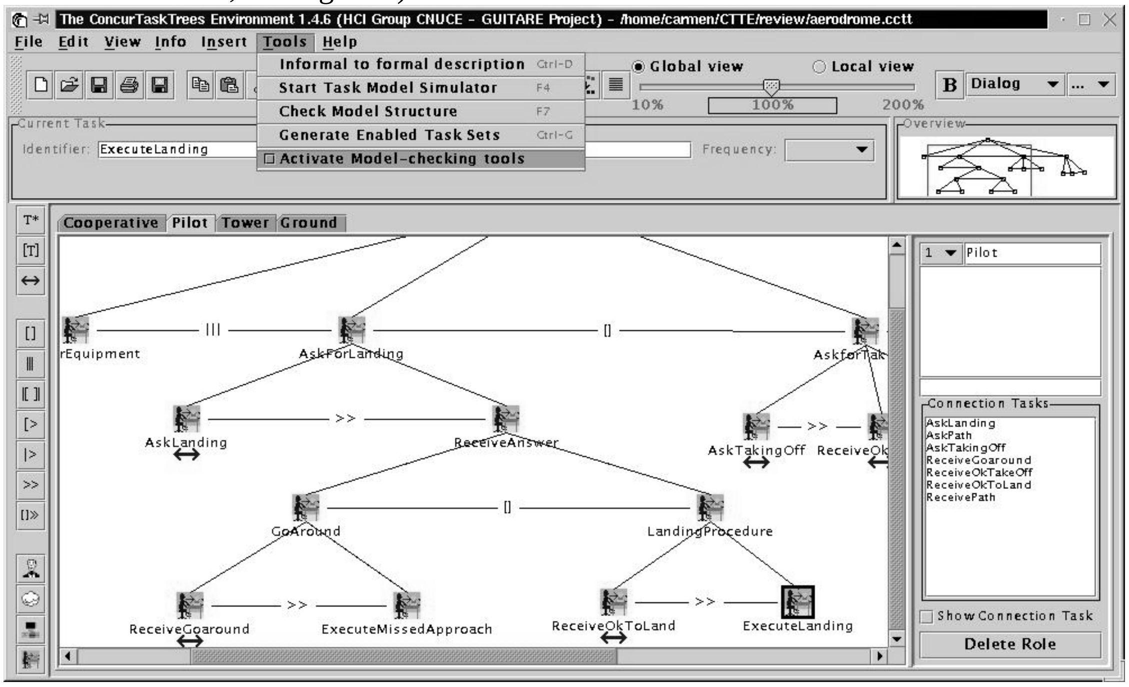
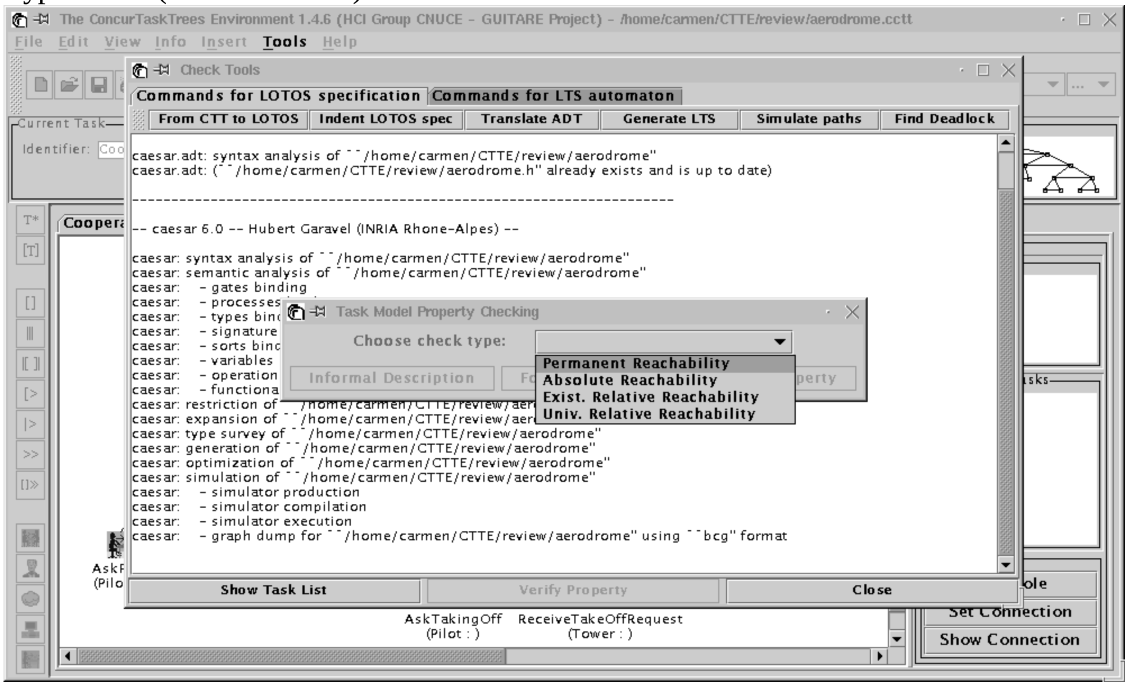


Figure 6: Activation of the model-checking tools

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 7). 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*).


The ConcurTaskTrees Environment 1.4.6 (HCI Group CNUCE - GUITARE Project) - /home/carmen/CTTE/review/aerodrome.cctt
File Edit View Info Insert Tools Help
Check Tools
Commands for LOTOS specification
Commands for LTS automaton
From CTT to LOTOS
Indent LOTOS spec
Translate ADT
Generate LTS
Simulate paths
Find Deadlock
Current Task
Identifier:
Task Model Property Checking
Choose check type:
Permanent Reachability
Absolute Reachability
Exist. Relative Reachability
Univ. Relative Reachability
Informal Description
Show Task List
Verify Property
Close
AskTakingOff (Pilot :)
ReceiveTakeOffRequest (Tower :)
Set Connection
Show Connection

```
caesar.adt: syntax analysis of ``/home/carmen/CTTE/review/aerodrome"
caesar.adt: (``/home/carmen/CTTE/review/aerodrome.h" already exists and is up to date)

----------------------------------------------------------------------

-- caesar 6.0 -- Hubert Garavel (INRIA Rhone-Alpes) --

caesar: syntax analysis of ``/home/carmen/CTTE/review/aerodrome"
caesar: semantic analysis of ``/home/carmen/CTTE/review/aerodrome"
caesar:    - gates binding
caesar:    - processes
caesar:    - types binc
caesar:    - signature
caesar:    - sorts binc
caesar:    - variables
caesar:    - operation
caesar:    - functiona
caesar: restriction of ``/home/carmen/CTTE/review/aer
caesar: expansion of ``/home/carmen/CTTE/review/aer
caesar: type survey of ``/home/carmen/CTTE/review/aerodrome"
caesar: generation of ``/home/carmen/CTTE/review/aerodrome"
caesar: optimization of ``/home/carmen/CTTE/review/aerodrome"
caesar: simulation of ``/home/carmen/CTTE/review/aerodrome"
caesar:    - simulator production
caesar:    - simulator compilation
caesar:    - simulator execution
caesar:    - graph dump for ``/home/carmen/CTTE/review/aerodrome" using ``bcg" format
```

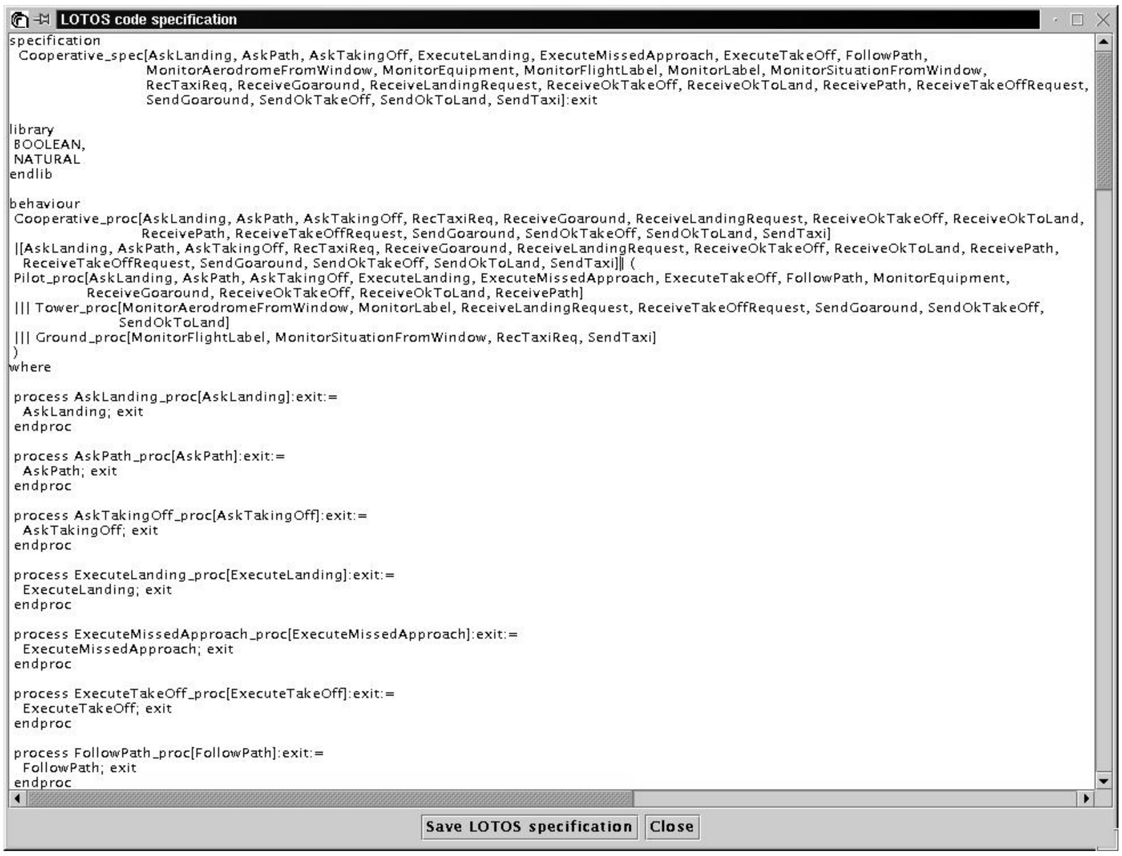
Figure 7: Selection of the type of property to check.

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"). The CTT environment will show a window where the corresponding LOTOS expression (see Figure 8) 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 9) 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, fig. 9) 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.



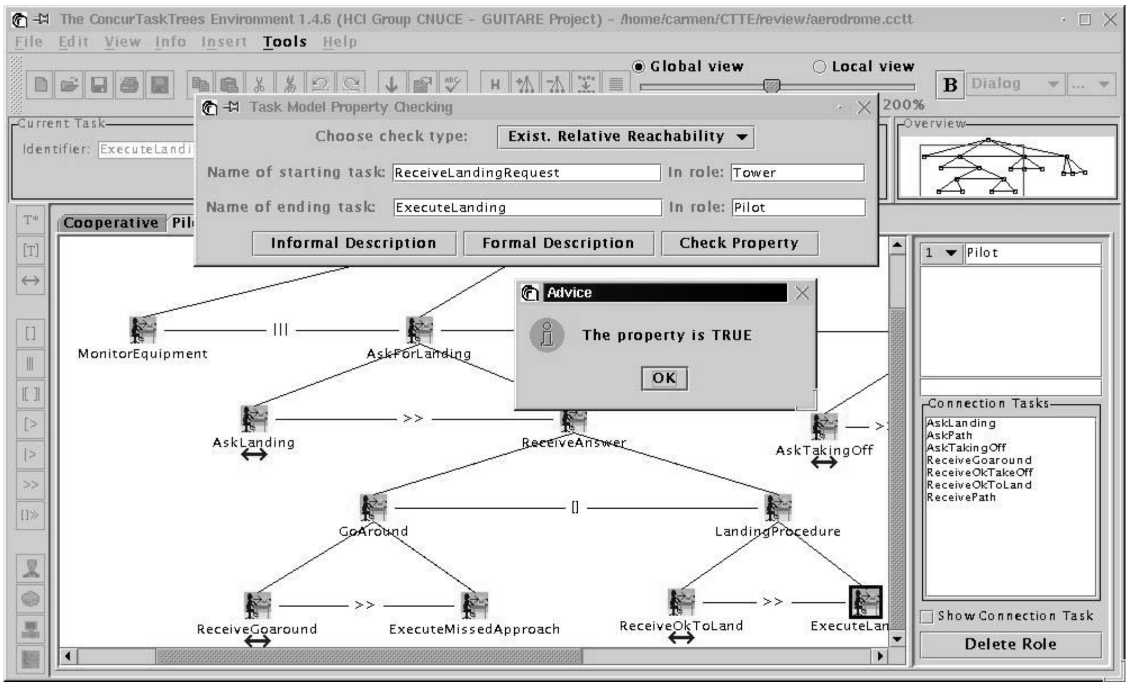
```
specification
 Cooperative_spec[AskLanding, AskPath, AskTakingOff, ExecuteLanding, ExecuteMissedApproach, ExecuteTakeOff, FollowPath,
                  MonitorAerodromeFromWindow, MonitorEquipment, MonitorFlightLabel, MonitorLabel, MonitorSituationFromWindow,
                  RecTaxiReq, ReceiveGoaround, ReceiveLandingRequest, ReceiveOkTakeOff, ReceiveOkToLand, ReceivePath, ReceiveTakeOffRequest,
                  SendGoaround, SendOkTakeOff, SendOkToLand, SendTaxi]:exit

library
 BOOLEAN,
 NATURAL
endlib

behaviour
 Cooperative_proc[AskLanding, AskPath, AskTakingOff, RecTaxiReq, ReceiveGoaround, ReceiveLandingRequest, ReceiveOkTakeOff, ReceiveOkToLand,
                  ReceivePath, ReceiveTakeOffRequest, SendGoaround, SendOkTakeOff, SendOkToLand, SendTaxi]
 |[AskLanding, AskPath, AskTakingOff, RecTaxiReq, ReceiveGoaround, ReceiveLandingRequest, ReceiveOkTakeOff, ReceiveOkToLand, ReceivePath,
  ReceiveTakeOffRequest, SendGoaround, SendOkTakeOff, SendOkToLand, SendTaxi]| (
 Pilot_proc[AskLanding, AskPath, AskTakingOff, ExecuteLanding, ExecuteMissedApproach, ExecuteTakeOff, FollowPath, MonitorEquipment,
           ReceiveGoaround, ReceiveOkTakeOff, ReceiveOkToLand, ReceivePath]
 ||| Tower_proc[MonitorAerodromeFromWindow, MonitorLabel, ReceiveLandingRequest, ReceiveTakeOffRequest, SendGoaround, SendOkTakeOff,
                SendOkToLand]
 ||| Ground_proc[MonitorFlightLabel, MonitorSituationFromWindow, RecTaxiReq, SendTaxi]
 )
where

 process AskLanding_proc[AskLanding]:exit:=
  AskLanding; exit
 endproc

 process AskPath_proc[AskPath]:exit:=
  AskPath; exit
 endproc

 process AskTakingOff_proc[AskTakingOff]:exit:=
  AskTakingOff; exit
 endproc

 process ExecuteLanding_proc[ExecuteLanding]:exit:=
  ExecuteLanding; exit
 endproc

 process ExecuteMissedApproach_proc[ExecuteMissedApproach]:exit:=
  ExecuteMissedApproach; exit
 endproc

 process ExecuteTakeOff_proc[ExecuteTakeOff]:exit:=
  ExecuteTakeOff; exit
 endproc

 process FollowPath_proc[FollowPath]:exit:=
  FollowPath; exit
 endproc
```

Figure 8: 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 9): 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 reveals flexibility because it allows users to specify in the same way both single-user and multi-user properties.



The ConcurTaskTrees Environment 1.4.6 (HCI Group CNUCE - GUITARE Project) - /home/carmen/CTTE/review/aerodrome.cctt

Task Model Property Checking

Choose check type: Exist. Relative Reachability

Name of starting task: ReceiveLandingRequest In role: Tower

Name of ending task: ExecuteLanding In role: Pilot

Advice

The property is TRUE

Connection Tasks:
AskLanding
AskPath
AskTakingOff
ReceiveGoaround
ReceiveOkTakeOff
ReceiveOkToLand
ReceivePath

Figure 9: An example of Property verification in the new environment.

9. An example of use

In this section we analyse an example drawn from our case study concerning the activity of air traffic controllers in control towers, in particular the management of the traffic in the proximity and within the airport.

In the situation considered, there are two types of controllers working elbow-to-elbow in the control tower: the *ground controller* and the *tower controller*. They communicate by means of the radio with pilots currently in the sector, by telephone with controllers of other centres and directly speak each other. For their activity, they mainly use the *paper flight strips*, which contain flight information (type of aircraft, planned route, etc.) and which they annotate to take into account the various flights' evolutions. 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 more detail:

- the *ground controller* has to look after movements "on the ground", which means (for departing flights) to guide planes from the departure gate until the site immediately before the runway's starting point (holding position) and (for arriving planes) from the end of the runway until the arrival gate.
- *tower controllers* have to take care of maintaining the minimal separation between aircraft then their activity is to allocate the access to the runway(s) and decide about take-off and landing of aircraft.

In our project, a new interactive prototype application for air traffic control in the aerodrome area was considered. It uses communication by data link, a technology allowing asynchronous exchanges of digital data containing messages coded according to a predefined syntax. In Figure 10 we show an excerpt of a ConcurTaskTrees task model specification of such prototype, describing the main activities performed by the tower controller in handling the traffic. 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 decide 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 control tower window (*CheckConflictsThroughWindow*), which involves performing the same activities as in the previous case if a conflicts is detected in the area, just in case (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 continuosly 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.

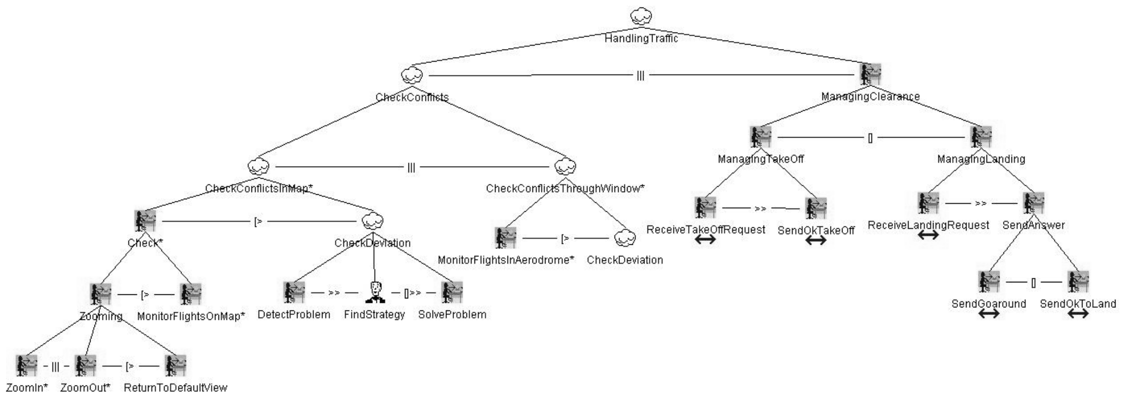


Figure 10: An excerpt of a ConcurTaskTrees specification

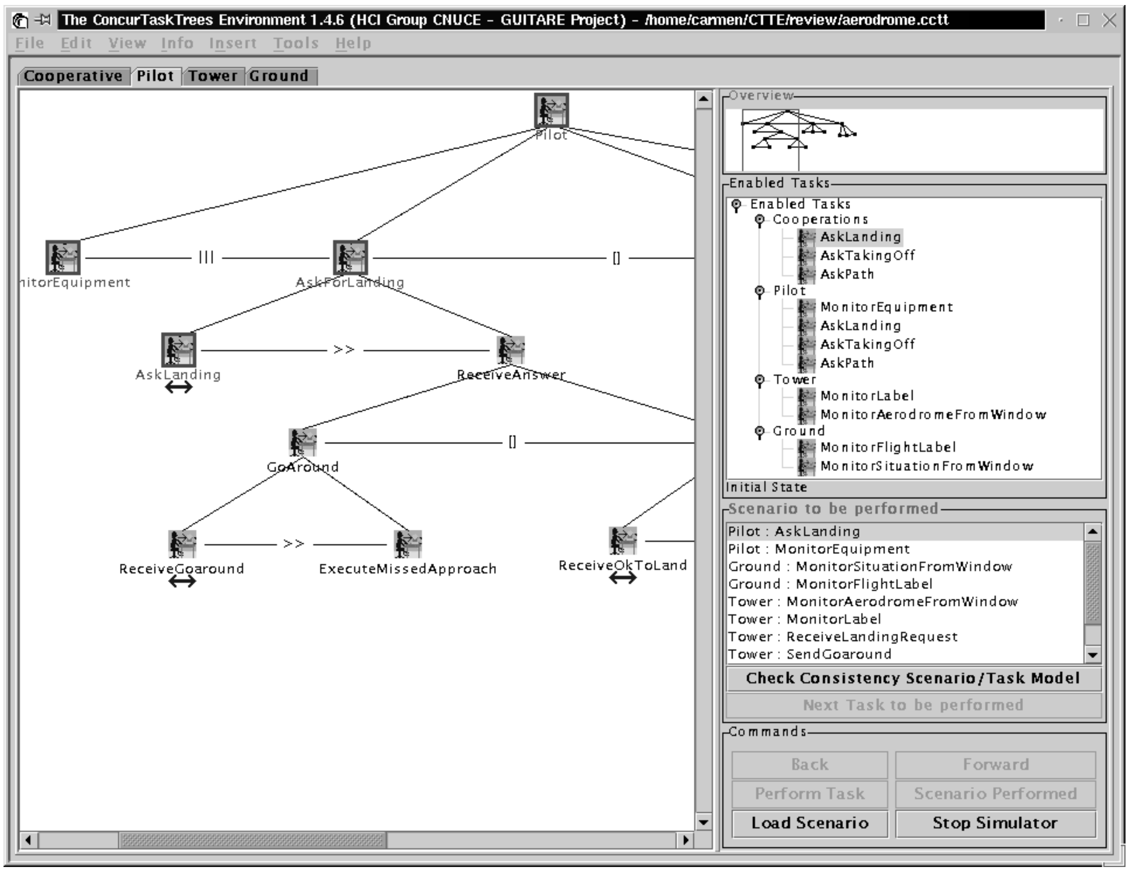
As we indicated before, the first step of our approach is to translate the CTT specification into a LOTOS specification, which is automatically performed. The second one is to specify a property that has to be checked against this model. For example, if the user wants to verify if it is possible for 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 9). 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 9, 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 11). 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.



Enabled Tasks
- Cooperations
  - AskLanding
  - AskTakingOff
  - AskPath
- Pilot
  - MonitorEquipment
  - AskLanding
  - AskTakingOff
  - AskPath
- Tower
  - MonitorLabel
  - MonitorAerodromeFromWindow
- Ground
  - MonitorFlightLabel
  - MonitorSituationFromWindow

```
Pilot : AskLanding
Pilot : MonitorEquipment
Ground : MonitorSituationFromWindow
Ground : MonitorFlightLabel
Tower : MonitorAerodromeFromWindow
Tower : MonitorLabel
Tower : ReceiveLandingRequest
Tower : SendGoaround
```

Figure 11: 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. Finally, our suggestion was to have the global view of the aerodrome map permanently displayed (at the default level of zooming) and, on request, to activate a separate window highlighting a specific part of the aerodrome (with a customised level of zooming).

During other exercises we were able to draw also other interesting results from the verification process, (for example we also proposed to introduce the possibility of a logical acknowledgement of any message sent to the pilots, in order to be sure that the message has correctly reached the pilot's system), which were the starting point for other useful discussions within the project.

Conclusions

In this paper we have presented and discussed a method that introduces the use of formal support in the design of interactive systems, particularly useful when interactive safety-critical applications are considered. We have explained how we build a task model of a cooperative application and then use it 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.
We have provided examples of the application of this method taken from a case study in the Air Traffic Control field: the management of aircraft in an aerodrome area with data link communications. Our method is supported by a set of tools (editing and analysis of task models, translator from ConcurTaskTrees to LOTOS, editor of formal properties of user interfaces) that can be integrated with existing model checking tools.
During the project, designers and developers of ATC system found the task model representations and the related environment understandable and useful, whereas they expressed doubts about being able to directly use model checking tools as they are, because their user interfaces and formal notations are too difficult and require too much effort. They also appreciated the possibilities supported by task modelling because it allows them to clarify a number of issues in very precise terms.
The tool for editing and analysing task models in ConcurTaskTrees is publicly available at http://giove.cnuce.cnr.it/ctte.html.
Future work will be dedicated to also supporting formal analysis of data values in this approach and the development of an agent-based temporal logic that can better exploit the features of the environment developed.

We gratefully acknowledge support from our colleagues from Alenia and ENAV in the MEFISTO Project (http://giove.cnuce.cnr.it/mefisto.html) for useful discussions.

References

1. Abowd, G., Wang, H., Monk, A. (1995) A formal technique for automated dialogue development. *Proceedings of DIS 95*, Ann Arbor, Michigan, 23-25 August, pp. 219-226. ACM Press.
2. Annet J., Duncan K.D. (1967) Task analysis and training design. *Occupational Psychology*, **41**, 211-221**.**
3. Breedvelt, I., Paternò, F., Severiins, C. (1997) Reusable structures in task models. *Proceedings of DSV-IS 97*, Granada, Spain, 4-6 June, pp. 251-265. Springer-Verlag.
4. Cairns, P., Jones, M. and Thimbleby, H. (1992) Reusable usability analysis with markov models. *ACM Transactions on Computer Human Interaction,* **8**(2), 99-132.
5. Campos J., Harrison M.D. (2001) Model checking interactor specifications. *Automated Software Engineering*, **8**, 275-310.
6. Campos, J.C., Harrison, M.D. (2000) The role of verification in interactive systems Design. *Proceedings of DSV-IS 98,* Abingdon, United Kingdom, 3-5 June, pp. 155-170. Springer Verlag.
7. Clarke, E.M., Emerson, E., and Sistla, A.P. (1986) Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems,* **8**(2), 244-263.
8. Constantine, L. & Lockwood, L.(1999) *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design*. Addison-Wesley, Reading..
9. Curzon P. and Blandford, A. (2001) Detecting multiple classes of user errors. *Proceedings of EHCI 01,* Toronto, Canada, 11-13 May, pp.57-72. Springer-Verlag.
10. D'Ausbourg, B., Seguin, C., Durrieu, G., Rochè, P. (1998) Helping the automated validation process of user interfaces systems. *Proceedings of ICSE 98,* Kyoto, Japan, 20-25 April, pp.219-228. ACM Press.
11. Del Bimbo, A., Vicario, E. (1999) A visual formalism for computational tree logic. *Journal of Visual Language and Computing*, **10**, 165-187.
12. De Nicola, R., Fantechi, A., Gnesi, S. and Ristori, G. (1993) An action-based framework for verifying logical and behavioural properties of concurrent systems. *Computer Network and ISDN systems*, **25**, 761-777.
13. Dix, A. (1991) *Formal Methods for Interactive Systems*. Academic Press.
14. Dix, A., Finlay, J., Abowd, G., Beale, R. (1998) *Human-Computer Interaction*. Prentice Hall Europe.
15. Emerson, E. A., Lei, C. L. (1986) Efficient model-checking in fragments of the Propositional mu-calculus. *Proceedings of 1st IEEE Symp. LICS,* Cambridge, Massachusetts, 16-18 June, pp. 267-278. IEEE Computer Society Press.
16. ITR-254 (2001) *An overview of CADP 2001*. Institut National de Recherche en Informatique et Automatique, Montbonnot Saint-Martin, France.
17. Garavel, H. and Sifakis, J. (1990) Compilation and verification of LOTOS Specifications. *Proceedings of PSTV 90,* Ottawa, Canada, 12-15 June, pp. 379-394. IFIP, North-Holland.

18. Johnson, C., and Botting, R. (1999) Reason's model of organisational accidents in formalising accident reports. *Cognition, Technology and Work*, **1**(3), 107-118.
19. Loer, K., Harrison, M. (2000) Formal interactive systems analysis and usability inspection methods: two incompatible worlds? *Proceedings of DSV-IS 2000*, Limerick, Ireland, 5-6 June, pp. 169-190. Springer Verlag.
20. Lusini, M., Vicario, E. (1998) Engineering the usability of visual formalisms: a case study in real time logics. *Proceedings of AVI '98*, L'Aquila, Italy, 25-29 May, pp. 114-123. ACM Press.
21. Mateescu, R. (1998) Verification des proprietes temporelles des programmes paralleles. PhD Thesis, Institut National Polytechnique de Grenoble, France.
22. Mori, G., Paternò, F., Santoro. C. (2002) CTTE: Support for Developing and Analysing Task Models for Interactive Systems Design. *IEEE Transactions on Software Engineering*, **28**(8), 797-813.
23. Palanque, P., Bastide, R. (1990) Petri Net with objects for the design, validation and prototyping of user-driven interfaces. *Proceedings of Interact 90,* Cambridge, UK, 27-31 August, pp.625-663. Elsevier Science, Cambridge.
24. Paternò, F. (1999) *Model-Based Design and Usability Evaluation of Interactive Applications*. Springer Verlag.
25. Paternò, F., Mancini, C., Meniconi, S. (1997) ConcurTaskTrees: A diagrammatic notation for specifying task models. *Proceedings of Interact 97,* Sidney, Australia, 14-18 July, pp.362-369. Chapman&Hall.
26. Paternò, F., Faconti, G. (1992) On the use of LOTOS to describe graphical interaction. *Proceedings of the HCI 92*, York, UK, 15-18 September, pp.155-173. Cambridge University Press.
27. Paternò, F., Mezzanotte, M. (1995) Formal verification of undesired behaviours in the CERD case study. *Proceedings of EHCI 95*, Grand Targhee Resort, Wyoming, 14-18 August, pp.213-226. Chapman & Hall.
28. Paternò, F., Santoro, C., Sabbatino, V. (2000) Using information in task models to support design of interactive safety-critical applications. *Proceedings of AVI 2000*, Palermo, Italy, 23-26 May, pp.120-127. ACM Press.
29. Rushby, J. (1999) Using model checking to help discover mode confusion and other automation surprises. *Proceedings of HESSD 99*, University of Liege, Belgium, 7-8 June.
30. Rushby, J. (2001) Modeling the human in human factors. *Proceedings of SAFECOMP 2001,* Budapest, Hungary, 25-28 September, pp 86-91. Springer-Verlag.
31. Thimbleby H. and Addison M. A. (1996) Intelligent adaptive assistance and its automatic generation. *Interacting with Computers*, **8**(1), 51-68.