Integrating Model Checking and HCI Tools to Help Designers Verifying User Interface Properties

Fabio Paternò, Carmen Santoro

Istituto CNUCE, Consiglio Nazionale delle Ricerche, Via V.Alfieri 1
56010 Ghezzano-Pisa, Italia
{F.Paterno, C.Santoro}@cnuce.cnr.it

Abstract. In this paper we present a method that aims to integrate the use of formal techniques in the design process of interactive applications, with particular attention to those applications where both usability and safety are main concerns. The method is supported by a set of tools. We will also discuss how the resulting environment can be helpful in reasoning about multi-user interactions using the task model of an interactive application. Examples are provided from a case study in the field of air traffic control.

1 Introduction

The importance of allowing an increasing number of users to access interactive software applications has stimulated a growing interest in methods for the design and development of effective user interfaces. The goal is to obtain applications able to support users, in a flexible and effective way, while they are performing their activities. Consequently, the part of an interactive software application dedicated to the control of dialogues with users and the generation of the presentation of the information is increasing thus raising the need for more structured methods to support its design. For example, Myers and Rosson [10] analyzed a set of applications and found that on average, 48% of the code, 45% of development time, 50% of implementation time, and 37% of maintenance time was dedicated to user interface aspects.

The fact that it is actually quite easy to develop one user interface by using one visual environment such as Visual Basic or Visual Java may lead to misconceptions. Thus, one question can be posed: why spend time investigating new methods for developing user interfaces if they are already so easy to develop?

The problem is that what designers of interactive applications need is not just any user interface able to provide access to the application's functionality but one that can effectively support users in interacting with it. Thus, there is a need to understand what tasks users want to perform, to represent them in order to better analyse their properties and interrelationships and to use this information to identify an effective user interface. Here is where most current visual environments for user interface development fail to provide useful support.

If we consider UML [4], we can notice a considerable effort to provide models and representations to support the design and development of software applications.

However, despite the nine representations provided by UML, none of them is particularly oriented to support the design of user interfaces. Of course, it is possible to use some of them to represent aspects related to the user interface but it is clear that this is not their main purpose.

Model-based environments for user interfaces [13, 15] is a research area that has stimulated an increasing interest. The basic idea is to identify abstractions and related tools that can support the work of user interface designers and developers. Particular attention has been paid to task models that are able to represent the design of activities that should be supported by an interactive application. Such models describe how activities should be performed and their possible relationships by integrating both functional and interactional aspects. They are the meeting point among all the important views involved in the design of an interactive application. Their development involves people with different backgrounds (software developers, user interface designers, application domain experts, end users, managers, …). Usually, such task models are developed after an informal phase of task analysis and scenarios identification.

One of the advantages of using a formal approach is the possibility to rigorously reason about properties of the specification. This can be carried out by model checking techniques: the specification represents the model against which properties can be checked. Formal verification has been successfully used in hardware design where it is important to check that some properties are satisfied before implementing the specification into hardware. 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 main problems in applying model checking techniques to the design of user interfaces are:

- the identification of relevant user interface properties to check and their formalisation;
- the development of a model of the User Interface System which is 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.

There are various motivations to carry out model checking for user interfaces:

- it is possible to test aspects of an application even if they have not been completely implemented;
- *user testing can be rather expensive*, especially in fields such as ATC (Air Traffic Control), the users (controllers in our case) are highly specialised personnel whose time has high costs. In addition, it can be very difficult to know how many tests are sufficient to have an exhaustive analysis. We are not proposing that users should not be involved in testing but we are indicating that model checking can decrease the need for empirical testing, even if it is always useful to have it.
- *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.

However, there is another difference between model checking and empirical testing: in the former a model of the application is considered, in the latter the focus is on the concrete implementation of the application (or part of it). This means that the model should be a meaningful approximation of the application in order to support a useful analysis.

In [12] there is a discussion on how to approach the verification of user interface properties and examples of general properties are given. Other approaches to the same type of problems can be found in [1, 2]. However, despite the number of proposals for the use of formal methods in HCI [11], only a small number of applications to real case studies has been developed (see [7] for one example). The main reason for these limited results is that the use of formal methods is difficult and time-consuming. Even tools supporting them are often difficult to use and the results that they can provide sometimes do not justify the effort required.

Our approach aims to analyse multi-users interactive applications and how it can be applied to a real case study where the effort of using a formal approach is justified by the safety-critical context. To this end we have designed and developed a prototype environment that integrates a tool for task modelling with a tool for model checking. The overall goal is to understand how to ease the introduction of formal methods in the design cycle and to understand to what extent formal techniques can be useful in designing interactive applications.

In the paper, we first introduce the method that we propose and we discuss the motivations and the results of each of its phases. Then, we describe the prototype environment that we have developed to support the integration of task modelling with model checking and the representations that it uses, and lastly we discuss examples of properties that can be verified with this approach. We also discuss examples taken from a case study that we have performed concerning the design of a new interactive application for air traffic control in an airport.

2 Our Method

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 represented formally. However, 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. Whereas in the case of user interface dialogues, there are aspects that are more difficult to grasp in an empirical analysis because when users navigate in an application they follow only one of the many possible paths of actions. In addition, interactive systems are highly concurrent systems because they can support the use of multiple interaction devices, they can be connected to multiple systems, they can support the performance of multiple tasks, they often can support multi-user interactions.

The current tendency is to increase such concurrency. On the one hand, this concurrency is a source of flexibility, usability, and interactional richness but, on the other hand, it generates a complexity that needs to be carefully considered, especially in safety-critical contexts. Thus, formal techniques can give a useful support to better understand dialogue models and their properties. Concerning when such a formalisation effort should be performed, we note that if only the lowest specification level of the task model (user actions and system feedback) was considered then many important aspects could be overlooked. A designer needs first to understand the logical and temporal relationships among the possible logical activities aspects (that to some extent depend on the artefacts available) and, consequently, a representation is required supporting such information. Formalising task models can allow designers to reach a number of results: a better understanding of logical activities to support; obtaining information useful for the concrete design of the user interface, the possibility of supporting usability evaluation of the application considered, the possibility of rigorously reasoning about properties. The last aspect has not sufficiently been considered 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 tasks and their performance.*

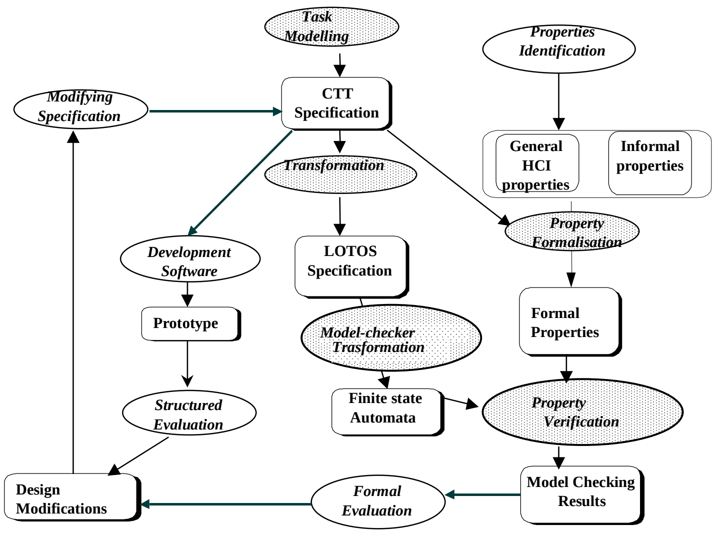


Fig. 1. Our Method

The method that we propose for the design of user interfaces with the support of formal methods is represented in Figure 1, 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 some of the tools involved in the method. We use model checkers developed by other groups [6] for the formal verification. 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 [13] which is a graphical notation allowing designers to describe hierarchies of tasks with temporal operators that have been identified extending the semantics of LOTOS [8] 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 that they manipulate and attributes such as frequency, time requested and so on. The editor of task models in ConcurTaskTrees is publicly available at (http://giove.cnuce.cnr.it/ctte.html).
- *ConcurTaskTrees-to-LOTOS transformation*. The ConcurTaskTrees specification can be used for two purposes. One is driving the development of a software prototype, in particular prototype presentation, consistent with the indicated requirements (for example, if we know that the application has to provide an overview of some information then presentation techniques suitable to summarise data should be considered). The other possibility is to transform the ConcurTaskTrees model into a LOTOS specification to analyse the dialogue of the user interface using model checking techniques. We have implemented a transformation tool where each task specification is translated into a corresponding LOTOS process. The motivation for this transformation is that there are various model checking tools able to accept LOTOS specifications as input. The LOTOS specification is the input for automatic tools (such as the CADP package [6]) that transform it into a finite state automata or Labelled Transition System (LTS). Since LOTOS has more expressive power than LTSs this transformation in some cases is not possible.
- *Property Formalisation*. The identification of the relevant properties of the user interface considers both general HCI properties, such as the continuous feedback property [12], and other properties that are specific of the considered application domain. To support user interface designers in editing formal properties we have defined a set of templates associated to 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.
- *Model-checking***.** After having formalised the identified properties with a formal notation, these properties are checked against the LTS specification, describing the Interactive System derived in previous steps, to verify which properties are valid in the system. Checking that the formal specification satisfies the relevant properties for the possible dialogues is useful to understand whether the design developed can support usability and safety aspects. The verification is performed by a general purpose automatic tool for formal verification and the results of the model checking are used for formal and informal evaluations that can lead to modify the ConcurTaskTrees specification thus re-starting the process.

3 The Case Study

The air traffic controllers' main task is to ensure flight safety and regularity: safety means that the minimal separation has to be maintained between aircraft, regularity means that the planes have to follow as much as possible the beforehand fixed flight plans. Currently, the air traffic controllers perform most of their activities using the following media and tools:

- Paper flight strips, automatically generated and printed by the system. Generally, there is one strip for each aircraft and each of them contains flight information (type of aircraft, planned route, etc.) and it is used by controllers to annotate the flight's evolutions in the sector;
- Vocal communications: controllers communicate via voice by means of radio with pilots currently in the sector, via telephone and directly with other colleagues working respectively in other and in the same center;
- Other instruments such as the radar that allows them to monitor the current traffic situation, especially used when it is not possible to have the complete overview of the traffic with the naked eye.

As far as it concerns the management of the traffic in the proximity and within the airport there are two particular types of controllers that work elbow-to-elbow in the control tower and who communicate with pilots using the radio with two different frequencies: the ground controller and the tower controller.

In the MEFISTO project, a new interactive prototype application for air traffic control in the aerodrome area has been designed and developed. It uses communication by data link, a technology allowing asynchronous exchanges of digital data containing messages coded according to a predefined syntax. This means that controllers can provide commands by direct manipulation, graphical user interfaces. This type of technology is particular useful in bad atmosphere conditions where the controllers have difficulties in observing the movements of aircraft from the control tower.

The *Ground* controller has to look after movements "on ground", which are (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. In order to perform their activity controllers have to mentally build the current picture of traffic and decide who, when and how can go through the taxiways (that allow movements from the various airport areas from/to the runways), and so on, minimising the likelihood of conflicts. More specifically, when pilots are approaching the runway, they inform the ground controllers, then the controllers send to them the frequency for contacting the tower controller, because at that point the flight passes under the tower controller's regulation.

Tower controllers have to take care of maintaining the minimal separations between aircraft thus their duty is to allocate the access to the runway and to decide the time of departure. They receive the strip from the ground controller and, when they receive the pilot's request for taking off they can send the relative clearance depending on the current situation. Thus, they have to mentally calculate how to manage the separations to ensure that no conflict could happen between departing planes and the effects of the "wake vortex" are cancelled.

4 Task Models

4.1 Representing Task Models

The reason for introducing ConcurTaskTrees was that after first experiences with LOTOS [14] we realised the need for a new extension that allowed designers to avoid unnecessarily complicated expressions even for specifying small behaviours and to focus on aspects more important for user interface design. In addition, we noted that other notations for task models were lacking of precise semantics or missing constructs useful for obtaining flexible descriptions.

The notation we use for representing task models allow designers to obtain a hierarchical description of the possible activities with a rich set of operators to describe their temporal relationships. Different icons are used to indicate how the performance of the task is allocated. A task model of a cooperative application is obtained by designing the task model associated with each role involved in an application and the model related to the cooperative part. In the latter part, cooperative tasks are considered, they are tasks that require actions from two or more users and they are refined until we reach tasks performed by a single user. This allows designers to define constraints among tasks performed by different users. An example of cooperative task is driving an aircraft to the holding position as it requires actions from the pilot, the tower controller and the ground controller.

In Figure 2 we show an excerpt of ConcurTaskTrees specification taken from our case study. It considers the *Taxi to* task performed by the Ground controller when he receives a path request from a pilot in order to reach either the assigned runway and then take-off or to leave the runway and reach the arrival gate. The controller must first select the corresponding interaction technique (*Select taxi to* task) and then (>> is the sequential operator) choose ([] operator) between using the path automatically suggested or building a path manually. In the former case, the controller selects a predefined path and the system shows it graphically (the relative icon indicates a system task). In the latter case, the controller selects manual mode. Then, an iterative task (*Building Path*, * is the iterative operator) allows specifying the next position and the system responds by graphically displaying the corresponding segment until the controller terminates the iterative building operation ([> is the disabling operator). Finally, the controller can choose between up-linking the path or cancelling it if he is not satisfied.

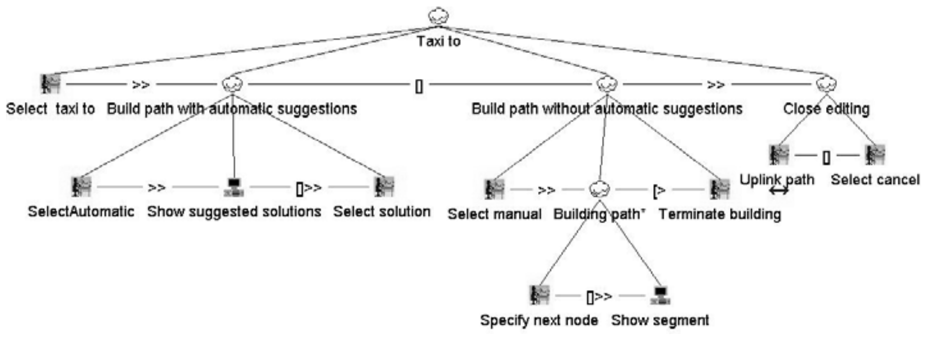


Fig. 2. An excerpt of a ConcurTaskTrees specification.

4.2 From ConcurTaskTrees to LOTOS

The first important aspect to consider is that the translation from ConcurTaskTrees to LOTOS is composed of two main steps: the translation of ConcurTaskTrees tasks into LOTOS processes, and to implement the ConcurTaskTrees temporal operators by means of the operators provided by the LOTOS language.

With regard to the former issue, for each process, its specification implies that all the gates that are used inside the specification have to be declared in its heading. The direct consequence is that in the specification of the root process all the gates detected have to be listed. Whereas in the specification of a process corresponding to a leaf task the translation is reduced to insert the associated gate and the *exit* action, in order to indicate when the successful end of the process occurs.

Some of the ConcurTaskTrees operators are derived from LOTOS, other operators (such as those for indicating optional and iterative tasks) need to be mapped onto LOTOS expressions. An example of the latter issue is in the translation of the ConcurTaskTrees iterative operator (*), that has not a direct correspondent in LOTOS. Referring to Figure 2, where the *Building Path* task is exactly iterative, its translation into LOTOS just 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 an its previous execution.

4.3 Integration of Tools for Task Modelling and Tools for Model-Checking

The environment that we have designed to support this method allows an integrated use of two tools:

- CTTE (ConcurTaskTrees Environment) [3] that supports editing and simulation of task models of cooperative applications.
- A model checking tool, we have used CADP [6] for this purpose but our environment can be easily integrated with other similar model checking tools.

In this case the designer can edit a task model and then automatically translate it into a LOTOS specification that is the input for the model checking tool. The integrated environment allows designers to directly access to some functionality of the model checker (*Activate Model-checking tools* item in the *Tools* menu, see Figure 3).

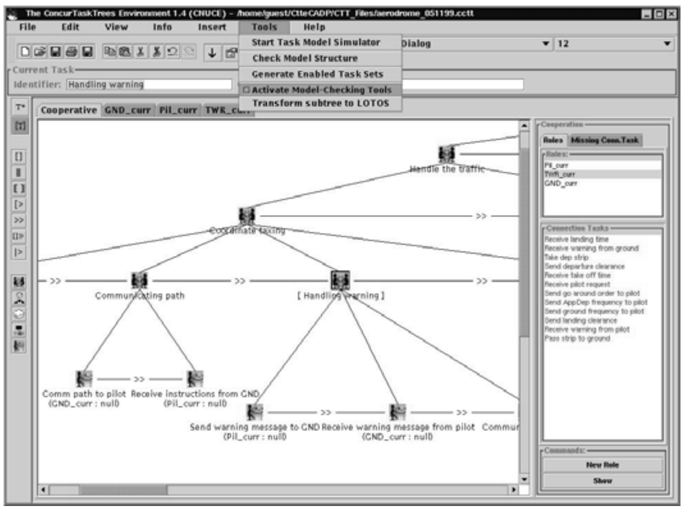
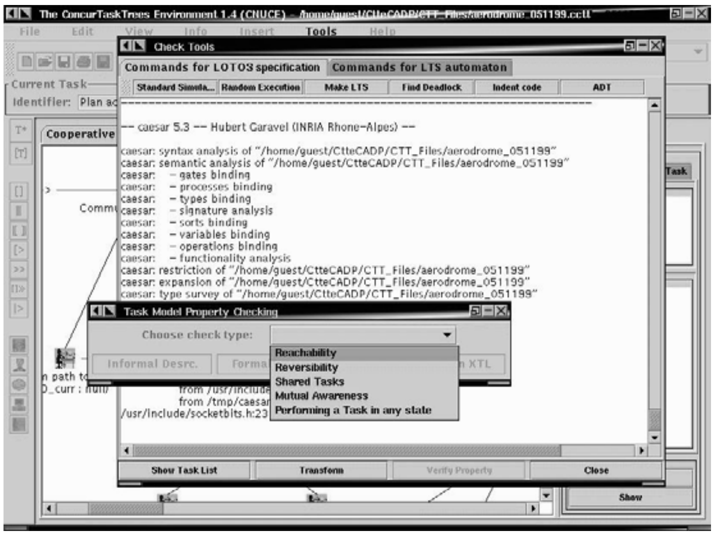

The ConcurTaskTrees Environment 1.4 (CNUCE) - /home/guest/CtteCADP/CTT_Files/aerodrome_051199.cctt
File
Edit
View
Info
Insert
Tools
Help
Start Task Model Simulator
Check Model Structure
Generate Enabled Task Sets
Activate Model-Checking Tools
Transform subtree to LOTOS
Dialog
12
Current Task
Identifier: Handling warning
Cooperative
GND_curr
Pil_curr
TWR_curr
Handle the traffic
Coordinate taxing
Communicating path
[Handling warning]
Comm path to pilot (GND_curr : null)
Receive instructions from GND (Pil_curr : null)
Send warning message to GND (Pil_curr : null)
Receive warning message from pilot (GND_curr : null)
Cooperation
Roles
Missing Conn.Task
Roles:
Pil_curr
TWR_curr
GND_curr
Connection Tasks
Receive landing time
Receive warning from ground
Take dep strip
Send departure clearance
Receive take off time
Receive pilot request
Send go around order to pilot
Send AppDep frequency to pilot
Send ground frequency to pilot
Send landing clearance
Receive warning from pilot
Pass strip to ground
Commands:
New Role
Show

Fig. 3. Activation of the model checking tools.

Then, a new window appears (see Figure 4) showing all the possible functionality that can be accessed from the tool. The new window is mainly divided into two panels: one includes commands used to get and handle the LOTOS specification, and the other one provides commands to handle Labelled Transition System (LTS) associated with the current task model. Thus, depending on the particular panel that has been selected, the window shows different sets of commands. For example, if the user selects the possibility to handle commands for the LOTOS specification then the tool gives various possibilities. They are (as you can see from the picture): performing a simulation of the LOTOS specification (*Standard Simula…*button); performing a casual *execution (Random execution*); producing the LTS (*Make LTS*) or finding deadlocks in the specification, if any, and so on.

In Figure 4 we show an example where the designer has activated the transformation into a LTS. After that, the editor of properties has been activated. In addition, through the *Transform* button appearing on the bottom of the *Check Tools* window, the designer can access further information, for example to get the LOTOS specification of the cooperative task model described in the CTT specification.

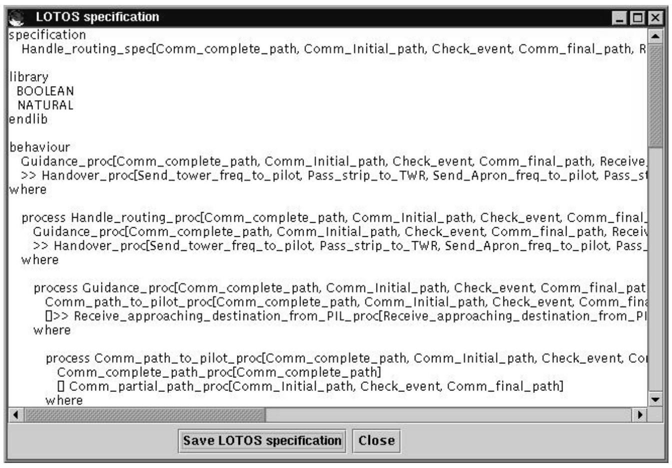


```
-------------------------------------------------------------------------

-- caesar 5.3 -- Hubert Garavel (INRIA Rhone-Alpes) --

caesar: syntax analysis of "/home/guest/CtteCADP/CTT_Files/aerodrome_051199"
caesar: semantic analysis of "/home/guest/CtteCADP/CTT_Files/aerodrome_051199"
caesar:    - gates binding
caesar:    - processes binding
caesar:    - types binding
caesar:    - signature analysis
caesar:    - sorts binding
caesar:    - variables binding
caesar:    - operations binding
caesar:    - functionality analysis
caesar: restriction of "/home/guest/CtteCADP/CTT_Files/aerodrome_051199"
caesar: expansion of "/home/guest/CtteCADP/CTT_Files/aerodrome_051199"
caesar: type survey of "/home/guest/CtteCADP/CTT_Files/aerodrome_051199"
         from /usr/include
         from /tmp/caesar
/usr/include/socketbits.h:23
```

Fig. 4. Selection of the type of property to check.

It is worth noting that, besides the possibility of having the entire task model translated into LOTOS, sometimes it could be useful to have the LOTOS expression correspondent to some subtasks. This can be obtained by selecting the *Transform subtree to LOTOS* item in the Tools menu (see Figure 3) after having selected in the editor the task name of the root of the subtree. For example, if we select in the editor window the *Building Path* task (represented in Figure 2), and then activate the translation into LOTOS, then the CTT environment shows a window where it is displayed the corresponding LOTOS expression (see Figure 5) that can be saved in a file.

In addition, to facilitate the specification of the properties the tool provides templates for predefined properties that can be filled interactively by selecting the tasks in the task model that should be involved in the property considered. The templates that have been selected allow designers to specify general properties such as relative reachability (verifying if it is possible to enable the performance of task y after having performed task x), performing a task in any state, mutual awareness, and so on. Depending on the specific property that the user wants to verify, the user interface provides an indication of the information that designers should fill. For example, if the user wants to verify if it is possible to perform a specific task in any state, then he should be able to specify exactly one task. Whereas if he wants to verify relative reachability he has to specify two tasks to check whether it is possible to perform the second task once the first one has been accomplished.

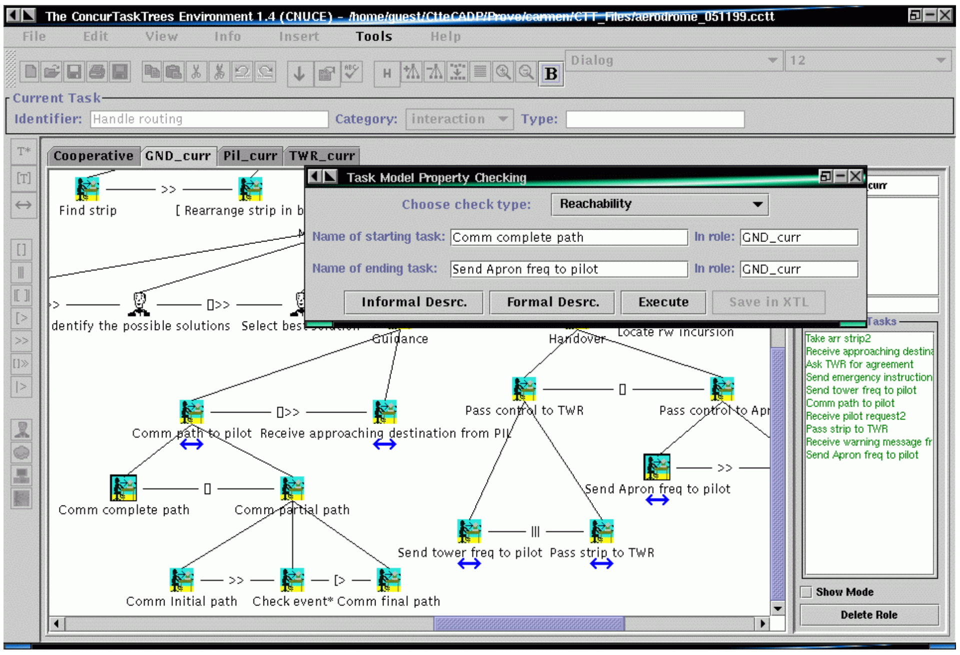


```
specification
  Handle_routing_spec[Comm_complete_path, Comm_Initial_path, Check_event, Comm_final_path, R

library
 BOOLEAN
 NATURAL
endlib

behaviour
  Guidance_proc[Comm_complete_path, Comm_Initial_path, Check_event, Comm_final_path, Receive
  >> Handover_proc[Send_tower_freq_to_pilot, Pass_strip_to_TWR, Send_Apron_freq_to_pilot, Pass_st
where

  process Handle_routing_proc[Comm_complete_path, Comm_Initial_path, Check_event, Comm_final_
    Guidance_proc[Comm_complete_path, Comm_Initial_path, Check_event, Comm_final_path, Receiv
    >> Handover_proc[Send_tower_freq_to_pilot, Pass_strip_to_TWR, Send_Apron_freq_to_pilot, Pass_
  where

    process Guidance_proc[Comm_complete_path, Comm_Initial_path, Check_event, Comm_final_pat
      Comm_path_to_pilot_proc[Comm_complete_path, Comm_Initial_path, Check_event, Comm_fina
      []>> Receive_approaching_destination_from_PIL_proc[Receive_approaching_destination_from_PI
    where

      process Comm_path_to_pilot_proc[Comm_complete_path, Comm_Initial_path, Check_event, Co
        Comm_complete_path_proc[Comm_complete_path]
        [] Comm_partial_path_proc[Comm_Initial_path, Check_event, Comm_final_path]
      where
```

Fig. 5. An example of LOTOS specification automatically derived.

Templates help designers identifying the desired property while preventing them from doing syntax errors that occur often in this kind of activity. For example, if the user wants to verify, within the Ground controller's task model, if it is possible to reach the Send Apron frequency to pilot task (*Send Apron freq to pilot)* from the Communicate complete path task *(Comm complete path*) then he has to select the *Reachability* item in the property list. After that, he has to graphically select in the task model the tasks involved in the property. In the case of reachability, two tasks have to be selected, so the left button of mouse allows designers to specify the first task and the right-button the second task: the associated fields are automatically filled, together with the correspondent roles (ground controller in the example of Figure 6).

The tool is able to provide the designer with both a formal and a natural language description of the property considered. If the designer decides to verify a property, then its formal specification is given to the underlying model-checking in order to verify if the specified property holds in the task model. In the negative case, the tool shows (one) execution that provides the 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.


The ConcurTaskTrees Environment 1.4 (CNUCE) - /home/guest/CtteCADP/Prove/carmen/CTT_Files/aerodrome_051199.cctt

Current Task

Identifier: Handle routing Category: interaction Type:

Task Model Property Checking

Choose check type: Reachability

Name of starting task: Comm complete path In role: GND_curr

Name of ending task: Send Apron freq to pilot In role: GND_curr

Informal Desrc. Formal Desrc. Execute Save in XTL

Take arr strip2
Receive approaching destina
Ask TWR for agreement
Send emergency instruction
Send tower freq to pilot
Comm path to pilot
Receive pilot request2
Pass strip to TWR
Receive warning message fr
Send Apron freq to pilot

Fig. 6. Interactive Composition of a Property.

5 User Interface Properties

In a highly cooperative system as that considered in our case study we can focus on properties of the specific user interface customised for a particular user and look at properties related to the interconnections and communications between different users. We can consider both general properties indicated in the pre-defined templates and specific properties of interest in the case study under consideration.

In multi-user properties we bring out other properties that regard characteristics of the highly cooperative environment considered, such as "awareness" properties which allow designers to reason about the possibility of checking if one user can be aware of the results of activities performed by another user. Within the "single-user" category it is possible to indicate the properties that are related to only one user indifferently from the particular user we consider because they are common to more than one user's class.

We will use an extended version of ACTL [5], which considers data values in addition to action, to formalise examples of properties in the case study previously introduced. This type of extension can be easily converted in an XTL [9] expression that can be verified by the CADP tool. Some properties are instances of the general properties that have specific templates associated with them, other properties are specific to our case study.

5.1 Examples of Single-User Properties

We start with one example of single user property and then show other examples involving multi-users interactions.

Warning Message for Time-Out Expired. With datalink functionality, all the messages have a time-out indicating the time interval during which the associated answer has to be received in order to be appropriately considered and evaluated. When the time-out expires, an appropriate notification has to be shown on the message originator's interface, in order to signal either that the message has to be sent again, or that possible answers received after the time-out expiration have to be ignored. More precisely, the property can be expressed in this way:

If there is expiration of an operational time-out without reception of the operational datalink response message, the message originator shall be notified with an appropriate feedback. The related ACTL-like expression is:

AG <is_sent(controller, request)> **E**[true{~is_received(controller, answer} **U** {timeout} **A**[true{true} **U** {is_presented(controller,noanswer_feedback}true]

This means that once the *controller* has sent a *request* to a *pilot*, then we have a temporal evolution during which no associated answer coming from the pilot to the controller has been received. Finally, as a result of the expiration of the fixed time-out, we reach a state from where for all the possible temporal evolution the desired effect of presenting an adequate feedback to the controller's user interface of the missed answer will be reached (*noanswer_feedback* in the above property).

This implies that only after the expiration of the time out we are sure that the desired effect (warning message for time-out expired) will be reached, thus allowing the controller to decide what is the best action to perform in order to make up for the error.

5.2 Examples of Multi-Users Properties

In this paragraph we consider some examples of properties of multi-user interactions that we found important to formalise for verification: awareness property, that in this case mainly means that an action produced by one controller has to be shown to the other sector's controller; co-ordination properties, and mutual exclusive properties.

Mutual Awareness Property. This property means that whenever a user performs an interaction, the associated effect has to be shown on the user interface of another user. We can use this property in our case study to be sure that the tower controller is aware of all the interactions (that we denote with “control_action” wording) performed by ground controllers which can have an impact on their activity. We consider both actions that controllers can perform directly on the system based on their decisions (for example the ground controller can change a previously fixed flight parameter), and actions that involve datalink dialogues with pilots. In other words, we want to pay attention to all the actions that might cause that controllers' activities clash each other, thus we do not consider the actions that the ground performs in order to get information on the system for monitoring it.

More precisely, we specify that whenever (AG operator) the ground controller performs a modification action on the user interface then for all the possible temporal evolutions (A operator) the event associated with the user interface modification reception will occur on the tower controller's user interface. The ACTL-like expression is:

AG<executed(ground, interaction)>**A**[true{true}**U**{presented(tower, feedback)}true])

With "executed" and "presented" wordings we want to distinguish when the system generates and undertakes the action from when the effects of the action are presented on the user interface. Of course, the property holds for the tower controller too. A direct consequence of the awareness is that the two controllers are more synchronised on these actions' sequences when (for example) a flight passes from one controller's handling to the other controller's. The most intuitive example is during the hand over from the ground controller to the tower controller for departure flights (whenever the pilot reaches the holding position, the ground controller performs a last contact and then the control is passed to the tower controller) and vice versa for arrival flights. In this case, the last contact message performed by the ground controller generates a feedback on the tower controller's user interface, so that the tower is aware of the performed action and he expects a pilot's message in the near future. In next figures we show an example of user interface in our prototype that supports this type of property.

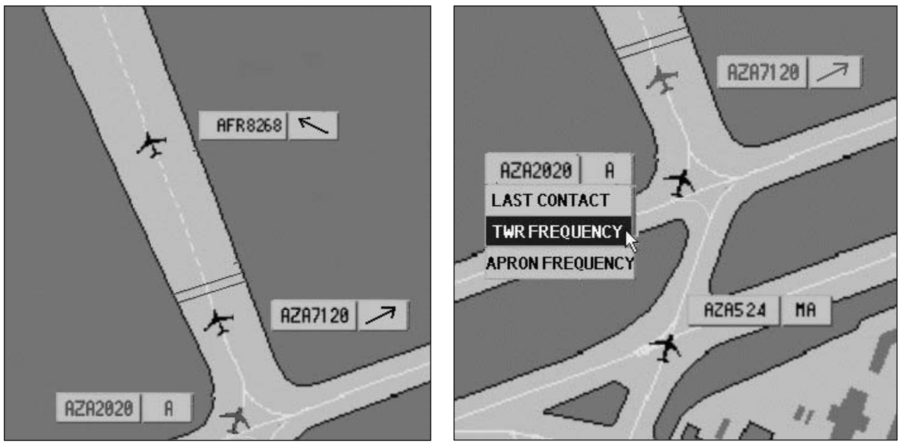


Fig. 7. User interface for tower controller (Left side), and for ground controller (Right side).

In Fig.7 (left-side) there is the Tower controller's user interface (as you can see its focus is mainly on the runway) at a certain time. The information about the flight AZA2020 is greyed because this flight is not currently under the control of the Tower controller. In Fig.7 (right-side) the user interface of the Ground controller is shown at the same time. As you can see, the Ground controller (whose activities are mainly dedicated to managing taxiways) is sending to the AZA2020 flight the frequency to contact the Tower controller. In Fig.8 the feedback of this action on the Tower controller's user interface is shown. The graphical technique that has been used to indicate that the Ground controller has sent the frequency of the tower controller to AZA2020 is an additional bold border around the flight label.

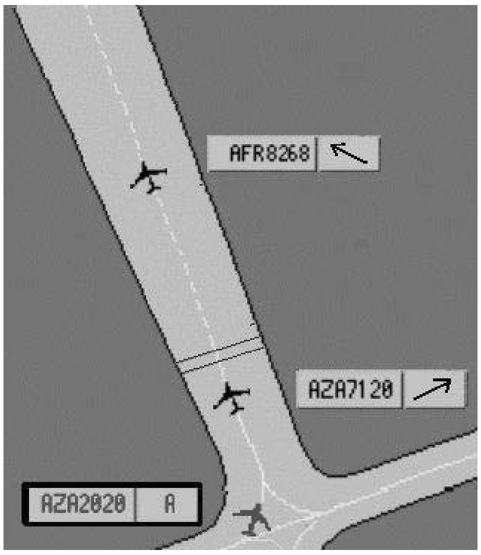


Fig. 8. Feedback for mutual awareness.

Location-Dependent Coordination Property. Here there is another case of proper controllers' *co-ordination.* In this case, different users are enabled to perform an interaction depending on the position of an object of interest: for example, this occurs when a departure flight has to cross an active runway in order to reach a different assigned runway. The ground controller gives to the flight a path on the taxiways until the flight reaches the runway that he has to cross. Thus, on the one hand the pilot knows that when he arrives at that point he has to wait for a message from a tower controller (who takes on responsibility for runways). In addition, more importantly, the tower controller knows that, when the pilot has reached the crossing he has to provide clearance to go through the runway as soon as possible, without any explicit request from the pilot:

AG<sent(ground, path)> **A**[true{true}**U**{received(pilot, path} **E**[true{true} **U**{stopped(pilot, runways_crossing)} **A**[true{true}**U**{sent(tower, ok_crossing} true]

This means that once the ground controller has sent the path to a pilot in order to reach the assigned runway, we have a temporal evolution during which the message has been received by the pilot. Then, we reach a state, by performing the pilot's action of stopping at the crossing of the taxiway with the runway, from where for all the possible temporal evolutions the desired effect (the tower controller sending the authorisation to cross the runway) will occur.

Mutual Exclusive Control Property. The tower and ground controllers share flight information of all the planes currently under their control (for example they can always obtain flight data by means of datalink menus). However, in order to serialise the control actions performed by each controller (for example sending datalink messages to pilots), it is important to guarantee that, while the flight is under the control of the tower controller, the ground controller can not send (voluntarily or unintentionally) control orders to pilot until the tower controller performs a last contact and then the flight passes under the control of the ground controller:

AG<sent(pilot, first_contact)> **A**[true{not (sent(ground, order) **U** {sent(tower, last_contact)} true]

This property considers when an arriving pilot sends a first contact message to the tower controller. Then, it will not be possible to have that the ground controller sends a control order to the pilot, until (**U** operator) the tower controller has sent to the pilot a last contact message.

6 Conclusions

In this paper we have presented and discussed a method that introduces the use of formal support in the design of interactive safety-critical applications. We have explained what the main aspects to consider in such formalisation efforts are and how we build a formal task model of a cooperative application that is then used to reason about single and multi-user properties. Such properties are identified through multidisciplinary discussions that involve end users, user interface designers, and software developers.

This approach has been applied to a case study in the Air Traffic Control field: the management of aircraft in the aerodrome area with data link communication. Our method is supported by a set of tools (editor of task models, translator from ConcurTaskTrees to LOTOS, editor of formal properties of user interfaces) that can be integrated with existing model checking tools.

Further work is planned on better integration between our tools and existing model checking tools in order to achieve, for example, more effective user interfaces for specifying properties and the possibility of analysing the results of the model checker directly in the ConcurTaskTrees model.

Acknowledgements

We gratefully acknowledge support from the European Commission for the MEFISTO Esprit LTR Project (http://giove.cnuce.cnr.it/mefisto.html) and our colleagues in the project for useful discussions on the topics of the paper.

References

1. B.d’Ausbourg, C.Seguin, G.Durrieu, P.Rochè, Helping the Automated Validation Process of User Interfaces Systems, Proceedings ICSE’98 pp.219-228.
2. G.Abowd, H.Wang, A.Monk, “A formal technique for automated dialogue development”, Proceedings DIS’95, ACM Press, pp.219-226.
3. G.Ballardin, C.Mancini, F.Paternò, Computer-Aided Analysis of Cooperative Applications, Proceedings Computer-Aided Design of User Interfaces, pp.257-270,, Kluwer, 1999.
4. G.Booch, J.Rumbaugh, I.Jacobson, *Unified Modeling Language Reference Manual,* Addison Wesley, 1999
5. R.De Nicola, A.Fantechi, S.Gnesi, and G.Ristori. An action-based framework for verifying logical and behavioural properties of concurrent systems, Computer Network and ISDN systems, 25, 1993, 761-778
6. H. Garavel, M. Jorgensen, R. Mateescu, Ch. Pecheur, M.Sighireanu, B.Vivien, CADP'97 - Status, Applications and Perspectives
7. A.Hall, “Using Formal Methods to Develop an ATC Information System”, IEEE Software, pp.66-76, March 1996.
8. ISO (1988). Information Processing Systems - Open Systems Interconnection – LOTOS - A Formal Description Based on Temporal Ordering of Observational Behaviour. ISO/IS 8807. ISO Central Secretariat.
9. R. Mateescu and H. Garavel, XTL: A Meta-Language and Tool for Temporal Logic Model-Checking. Proceedings of the International Workshop on Software Tools for Technology. Transfer STTT'98 (Aalborg, Denmark), July 1998.
10. Myers, B., Rosson, M.B., “Survey on User Interface Programming”, *Proceedings CHI’92*, pp. 195-202, ACM Press, 1992.
11. P.Palanque, F.Paternò (eds.), Formal Methods in Human-Computer Interaction, Springer Verlag, 1997.
12. F.Paternò, Formal Reasoning about Dialogue Properties with Automatic Support, Interacting with Computers, August 1997, pp.173-196, Elsevier.
13. F.Paternò, Model-Based Design and Usability Evaluation of Interactive Applications, Springer Verlag, ISBN 1-85233-155-0, 1999.
14. F.Paterno', G.Faconti, On the Use of LOTOS to Describe Graphical Interaction, in Monk, Diaper & Harrison eds. People and Computers VII: Proceedings of the HCI'92 Conference, pp.155-173, Cambridge University Press.
15. A.Puerta, A Model-Based Interface Development Environment, *IEEE Software*, pp. 40-47, July/August 1997.