Goal-Oriented Design of Domain Control Panels

Christophe Ponsard[1], Nadiya Balych[2], Philippe Massonet[1],
Jean Vanderdonckt[2], and Axel van Lamsweerde[2]

[1] CETIC Research Center, Charleroi, Belgium
[2] Université Catholique de Louvain, Belgium

Abstract. Goal-oriented methodologies have demonstrated some adequacy for modelling composite systems, from high level desired properties to operational requirements on responsible agents. This paper shows how to derive a user interface for human agents from such a model, especially with respect to the monitor and control capabilities of those agents. A goal-oriented widget taxonomy was elaborated in order to facilitate selecting widgets that are appropriate for each element of the underlying domain model. A user-friendly tool for building user interfaces, supporting the retrieval of adequate components and their fine tuning at a graphical level, was developed and deployed on the animator of the Objectiver/FAUST requirements toolbox.

1 Introduction

For many years, Model-Based Interface Development Environments (MB-IDEs) have been largely driven and inspired by the same suite of typical models such as: task, domain, user, abstract user interface, concrete user interface, final user interface, platform, etc. [17][2]. Among these, two classic models typically initiate the development process: the domain and task models. The former has been largely used during the past decade to automatically generate a graphical user interface. The later was defined to address some shortcomings. Today, it represents the most common model that drives the development process to foster user-centered design. These models have demonstrated some relevance, some convenience, and some efficiency in producing user interfaces for a very specific type of interactive system: information systems, where the User Interface (UI) mainly consists of a predefinition of forms, windows, menu bars, pull-down menus, etc. Typical information systems could be developed in Interface Builders such as Microsoft Visual Basic, C++, Java or Web applications. Traditional MB-IDEs have demonstrated the feasibility of the approach for such systems.

But when it comes to developing the UI of a complex, reliable system that is not an information system (e.g., a control system, a simulator), in particular when the UI cannot consist of predefined widgets with a predefined behavior, it seems more difficult to reuse the same suite of models. Since in this case these models have not been expanded enough to capture sufficient information to initiate the development process, perhaps other models need to be used.

In addition, the task model is widely recognized and used in the Human-Computer Interaction (HCI) community, whereas the Software Engineering (SE) and more specifically Requirements Engineering (RE) community, tends to ignore such a model and

S.W. Gilroy and M.D. Harrison (Eds.): DSVIS 2005, LNCS 3941, pp. 249–260, 2006.
© Springer-Verlag Berlin Heidelberg 2006

rather prefer to exploit models that are more traditionally used in their field. This paper investigates whether the shortcomings of most MB-IDEs for complex, reliable, time constrained interactive systems could be developed following a model-based approach, but based on another suite of models than those which are traditionally used in HCI.

In particular, the task model usually represents in HCI the user's viewpoint by recursively decomposing the task into sub-tasks that are connected with temporal operators to end up with leaf tasks. Of course, each task is associated with one or several goals that need to be achieved when the task has been carried out. In contrast, in RE, a goal model represents the system's viewpoint by specifying a decomposition of goals into sub-goals that are linked together with several operators to end up with leaf goals assigned to an agent.

As in HCI, the initial goal model does not come alone. It is then related to three other models that progressively capture relevant information for the other aspects: object model, agent model, and operation model (Sect. 2). These models are frequently used in the area of RE which provides capabilities to reason about human agent interface design with respect to the requirements under its responsibility (Sect. 3). Therefore, when it comes to specifying the components of the final user interfaces, i.e. the widgets to be used, instead of having a widget classification that is based on the task and/or the domain, the widget ontology should also be revisited. It should provide a rich framework to select the most appropriate combination of displays/controls for each agent to realize its goals (Sect. 4). Then, an animator should provide agent-oriented animations in the same way as the task can be simulated in traditional MB-IDEs [14][1] (Sect. 5). Finally, related work will be reviewed to highlight the contribution and limitations of our approach.

Throughout this paper, a non-trivial train system will be used as running example. Although simplified for fitting the size constraints of this paper, this system is based on implementations taken from real world railway signaling systems.

2 Background: Goal-Oriented Modeling

This paper considers goal-oriented modelling with the KAOS language which is organised in 4 models: (i) the central model is the *goal model* which captures and structures the assumed and required properties; (ii) the *object model* captures the relevant vocabulary to express the goals; (iii) the *agent model* takes care of assigning goals to agents in a realizable way; (iv) the *operation model* details, at state transitions level, the work an agent has to perform to reach the goals he is responsible for.

2.1 Building the Goal and the Object Models

Although the process of building those 4 models is intertwined, the starting point is usually a number of key properties of the system to-be. Those are expressed using *goals* which are statements of intent about some system (existing or to-be) whose satisfaction in general requires the cooperation of some of the agents forming that system. *Agents* are active components, such as humans, devices, legacy software or software-to-be components, that play some role towards goal satisfaction. Some agents thus define

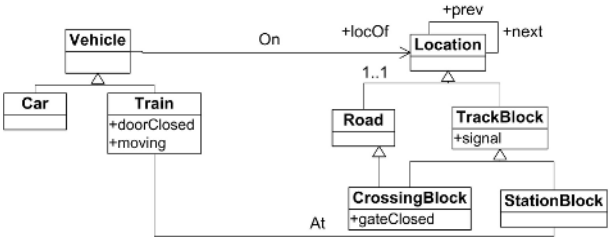

Fig. 1. Object Model

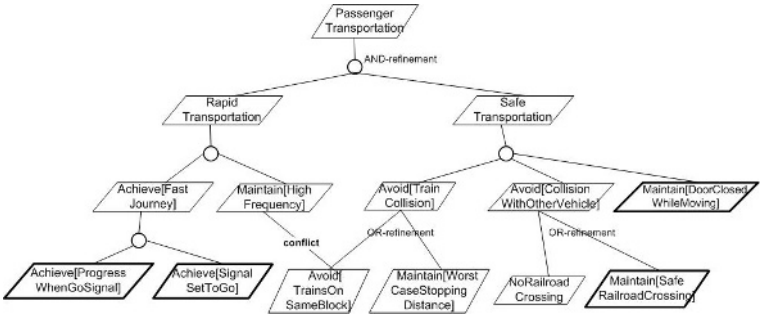

Fig. 2. Portion of the Goal graph

the software whereas the others define its environment. Goals may refer to services to be provided (functional goals) or to the quality of service (non-functional goals). Goals are described informally in natural language (InformalDef) and are optionally formalized in a real-time temporal logic (FormalDef) [3][8][12].

In our example, the goals relate to safety of transportation: avoiding collision between trains, with other vehicles (especially at railroad crossings), securing passenger (eg., by closing doors when moving). This goal can be stated as follows:

Goal Maintain[DoorsClosedWhileMoving]
InformalDef: Doors should be closed when the train is moving.
FormalDef: $(\forall tr : Train)\ tr.moving \Rightarrow tr.doorClosed$

In the above formulation, we have identified the *Train* entity with its *moving* and *doorClosed* attributes. Those are incrementally added to the structural model which captures passive (entities, relationships and events) and active objects (agents).

Unlike goals, *domain properties* are descriptive statements about the environment, such as physical laws, organizational norms or policies, etc. (eg., a train requires some distance to come to stop—law of inertia).

A key characteristic of goal-oriented requirements engineering is that *goals* are structured and that guidance is provided to discover that structure and refine it until agents can be found to realize those goals in cooperation. In KAOS, *goals* are organized in AND/OR refinement-abstraction hierarchies where higher-level goals are in general

strategic, coarse-grained and involve multiple agents whereas lower-level goals are in general technical, fine-grained and involve fewer agents [4]. In such structures, *AND-refinement* links relate a goal to a set of subgoals (called refinement) possibly conjoined with *domain properties*; this means that satisfying all subgoals in the refinement is a sufficient condition in the domain for satisfying the goal. *OR-refinement* links may relate a goal to a set of alternative refinements.

Figure 2 shows the goal structure for our system. It was set up starting from a few initial goals and by asking respectively "WHY" and "HOW" questions to discover parent goals (such as *Maintain[SafeTransportation]*) and son goals (such as *Maintain[DoorClosedWhileMoving]*).

2.2 The Agent Model

Goal refinement ends when every subgoal is realizable by some individual $agent$ assigned to it, that is, expressible in terms of conditions that are monitorable and controllable by the agent [10]. A *requirement* is a terminal goal under responsibility of an agent in the software-to-be; an *expectation* is a terminal goal under responsibility of an agent in the environment. For example:

Agent TrainDriver
Kind: Human
ResponsibleFor: Maintain[DoorClosedWhileMoving], Achieve[TrainProgress...
Monitors: $Block.signal$
Controls: $Train.moving, Train.doorClosed, Train.blockOf$

In our design, all agents are part of the automated system except the train driver, a human which stays in control of the moving state and the location of the train. The *door and motor controllers* are running on board of the train, while the *signal controller* and the *gate controller* are ground-based. In our case study, a very important problem to reason about is the way to relay ground and board information w.r.t their monitor/control needs. Figure 3 shows the *agent interface view* which displays the flow of monitored/controlled information among agents.

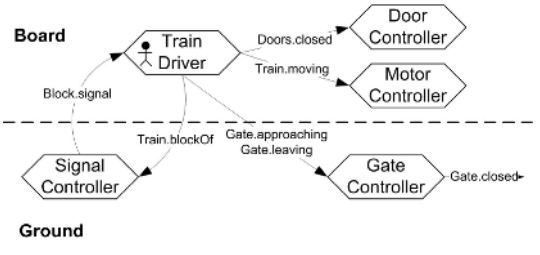

Fig. 3. Agent Interface Model

2.3 The Operation Model

Goals are operationalized into specifications of operations to achieve them [3]. An $operation$ is an input-output relation over objects; operation applications define state

transitions along the behaviors prescribed by the goal model. The specification of an operation is classical with precondition (necessary), postcondition (target state) and trigger condition (sufficient). An important distinction is also made between (descriptive) domain pre/postconditions and (prescriptive) pre-, post- and trigger conditions required for achieving some underlying goal(s). For example, the $OpenDoors$ operation may be specified as follows:

Operation OpenDoors
Input: tr:Train/doorClosed
Output: tr:Train/doorClosed
DomPre: $tr.doorClosed$
DomPost: $\neg tr.doorClosed$
ReqPre: for $SafeRailroadCrossing$ $\neg tr.moving$
ReqTrig: for $FastPassengerTransfert$ $(\exists st : Station)$ $at(tr, st)$

A goal operationalization is a set of such specifications. For example, to be complete, our operationalization of the goal $DoorClosedWhileMoving$ would also require to strengthen the *StartTrain* operation.

3 Overview of the Panel Design Process

Designing a control panel from a correctly refined and operationalized model can be done in a systematic way as the model, and more specifically the agent sub-model, captures all the relevant information. The overall process is the following:

- For each human agent in the goal model, retrieve all the requirements under his responsibility
- For each human agent in the agent model, identify all the information he has to monitor and to control w.r.t the realization of those requirements.
- For each human agent in the in the operation model, identify the operation(s) under the performance of that agent and through which he may affect controlled elements.
- For each monitored element, select an appropriate *display* widget and map it on the monitored element.
- For each controlled element, select an appropriate *control widget* and map it on the related operation(s).

In our example, the *TrainDriver* is responsible for the goal *Maintain [DoorClosedWhileMoving]*. The agent model shows he has to control both the *doorClosed* state and the *moving* state. The $TrainDriver$ is also responsible for other goals such as *Maintain[ProgressWhenGoSignal]* and *Maintain[StopOnRedSignal]* which requires the ability to monitor *Signal* and again to control the *moving* state. For the controlled part: *doorClosed* and *moving* are boolean attributes whose control is achieved through the *Start/Stop* and *OpenDoors/CloseDoors* pair of operations. In this case, an adequate control widget is a two-state switch for which each transition has a corresponding operation in the model. It also provides feedback on the controlled state.

As human agents are limited in many ways, it is important to check for the effective monitorability/controlability of the considered information. *Lack of monitorability* may be caused by limitation in perception capabilities (eg. danger of missing/misinterpreting

track information at high speed), limited focus (eg. looking at track signals and on-train controls), etc. *Lack of controllability* may be caused by limitation in response time (eg. emergency braking at high speed), high load, etc. Those specific properties can be taken into account when reasoning about the realizability of the design of the system using a number of available techniques (obstacle[20], agents tactics[10]). For example, when reasoning about the monitorability of block occupation, direct physical perception of track signals is only possible at low speed. High speed would be an obstacle which can be resolved by a more elaborate system design, with in-cabin reporting and relying on ground-cabin communications. *Human agents can also forget things and make mistakes*: the KAOS framework can model and reason about this using knows/belief predicates in goal formulations. The reasoning can be done directly, at refinement time, or more preferably later, using obstacle analysis[20] performed on an ideal model to evaluate possible consequences and mitigate them.

4 Building the Widget Taxonomy

In order to facilitate the design of interactive panels with respect to the underlying goal model, several real-world control panels were studied in a number of domains such as transportation, industrial control and domestic use. They helped in the definition of a widget taxonomy in terms of goals (being able to monitor/control state, to report dangers, etc.) instead of directly being focused on data or tasks. So far, several widget taxonomies have been investigated, but based on other models such as domain model [21] or a task model [13]. In those existing taxonomies, widgets are classified according to the underlying data or domain model they can manipulate, but do not differ whether the data are involved as input, output, or w.r.t their impact on system goal. Rather, in the present taxonomy, it is possible to directly map the previously identified goals and sub-goals (especially the leaf goals) to the widget they can best represent.

The resulting taxonomy is partially shown in Fig. 4. It is structured much in the same spirit as a KAOS model: the first level of the classification is a generic model capturing the goals the UI should address. Each goal is refined until relevant widget classes are identified. Those are then structured based on a more traditional data type hierarchy. At the bottom of the figure, some typical instances are illustrated.

In the resulting taxonomy, a number of *conceptual attributes* were identified such as display/control/alarm kind (goal level) and the type of data shown (data level). Based on the study of several widget instances, a number of *graphical attributes*, not part of the taxonomy, were also identified: those help in tuning the visual aspect: layout, divisions, labels, units, etc. For the speedometer dial shown on the left of Fig. 5, conceptual attributes are: display modality, continuous data range; graphical attributes are: angular layout, subdivisions, unit. All those features are captured by a single concept which is the “computer version” of that widget and of other “dial” widgets sharing the same conceptual attributes. It is implemented by a corresponding class. In this work, the Java language was used together with a JavaBeans interfacing mechanism and the Swing graphical toolkit.

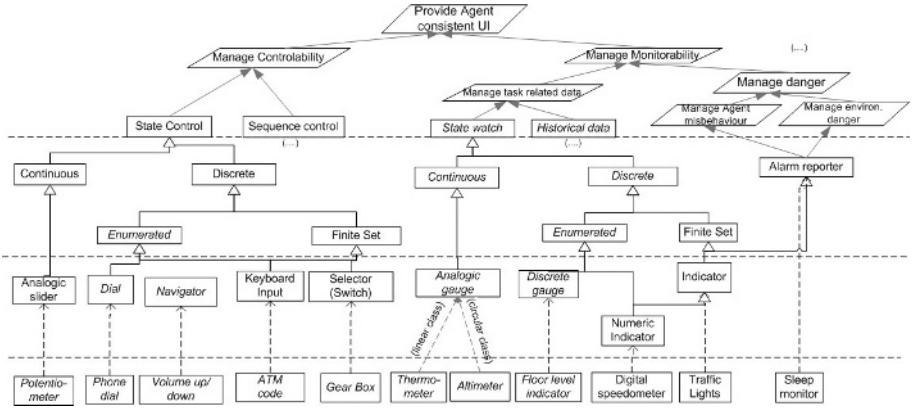

Fig. 4. Widget Taxonomy

Fig. 5. Physical (left) and modeled (right) dials from various domains

5 Deployment on a Requirements Animator

An important aspect of using models from the beginning of the development process, especially for complex systems like interactive monitoring and supervision systems, is the capability to execute the elicited models in some way to animate a first version of the future system for validation purpose. So, it is crucial that the underlying models are expressive enough to obtain a running interface. For instance, the Visual Task Model Builder [1] mainly exploits the task model to animate the ordering of tasks in time. This is certainly interesting but is not detailed enough to obtain a working prototype that is executable enough to demonstrate non-trivial interaction sessions. Similarly in CTTE [14], it is possible to animate the task model to see possible orderings of tasks in time, but the task model is not expressive enough to demonstrate a complete executing of a task. For this purpose, we wanted to exploit all underlying models as exposed in Section 2 to obtain an animator that is really working without coding or modeling more aspects.

The FAUST requirements animator supports the animation of KAOS goal-oriented models [19], [16], [18]. Relying on the Objectiver RE platform[15], the FAUST animator helps in the design of the system model, by the ability to generate and replay execution traces and in the validation, by allowing domain experts to interact with an executable model using a familiar UI. The existing animator supports the compilation of operationalized goal-models into finite state machines (FSM) which are then run in a simulator engine on a server. Several client actors can then connect on that simulator to

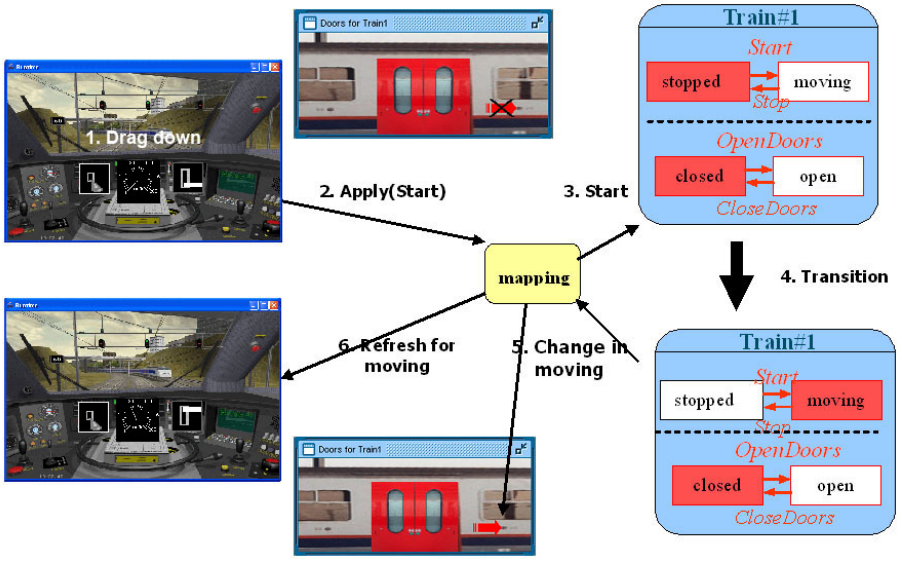

Fig. 6. An Animation Scenario

interact with the model in a multi-user way, enabling interesting validation scenarios to be played. The animator is complemented by a monitor which reports goal violations.

Presently, the animator suffers from a three important shortcomings: (1) the lack of control using domain specific interfaces, (2) the lack of consistent agent-based UI (currently an animator actor is not specially bound to an agent in the model, it may not give enough monitor/control or too much) and (3) the lack of a support for the development of such interfaces.

The first and second points were addressed in the previous sections. On the tool side, the composition and mapping is supported by a visual tool allowing the animation designer to assemble the panel by performing queries on the widget library for mapping each relevant variable. The tool was implemented in Java with the Swing graphical library. It supports composition by drag-and-drop of components on the panel. A JavaBeans based property editor is available for instantiating the available widgets and tuning their graphical rendering. A number of common widgets are also proposed for domains such as automotive, railroad, aerospace and industrial control.

The integration within the animator is achieved using a mapping component which manages the consistency between the underlying conceptual model (ie. states and transitions of the generated FSM) and the UI (controls and displays) of the animation. To achieve maximal flexibility, it can be defined at type level and tuned at instance level. The mapping is fully described in [19] and can be roughly summarized as follows:

– *Display widgets* are mapped on state elements (which are also linked to instances of the object model). The type of the widget and the underlying state should be compatible.

– *Control widgets* are mapped on an operation which is applied when a user action is performed. The performer is the human agent associated with the enclosing panel. Operation arguments are derived from the input capabilities of the widget (eg., rate of acceleration) and from other observable states (eg., current speed used as set point for a cruise control). Additional constraints can also be expressed using instance information (eg., speed applies to train controlled by the agent).

Figure 6 shows a typical animation scenario in two steps, respectively at top and bottom of the figure. The control panel, the lateral view of the train and the FSM instances are displayed from left to right. The user playing the ”driver” role makes a request to start the train (1). It is forwarded to the simulator (2,3) which accepts it (the guard being enabled because doors are closed). After the resulting transition (4), the train is now moving and a state refresh is triggered on the control panel (speedometer) and the lateral view (“forward” arrow) (5,6).

In the above scenario, note that if the doors were opened when requesting the start, that user action would have been rejected. This could also be signalled at the UI level using information widgets. This is an interesting feature for modelling automated backup mechanisms behind a human agent.

6 Related Work

FAUST and its animator, as outlined in this paper, mainly differs from existing MB-IDE approaches in the sense that four models are borrowed from requirements engineering. Therefore, these models address functional (FR) and non-functional requirements (NFR) in the different models as opposed to solely some NFR in task-based approaches. For instance, it is possible in FAUST to automatically execute the underlying models to obtain a genuine dialog whereas in traditional MB-IDEs, this dialog model should be generated from the task model, which represents a complex task. Furthermore, FAUST is based on all types of requirements, whereas traditional MB-IDEs mainly focus on user's viewpoint and requirements, such as portability, usability, etc. They decompose the task into sub-tasks, but without considering seriously FR such as domain constraints, robustness, reliability, performance, avoiding of conflict. Such properties are automatically obtained by construction and by generation in the FAUST process, whereas they usually need to be done by hand in MB-IDEs, based on the existing models [21].

Most animations exploit the task model [1] [13], whereas FAUST exploits all four models. Other animation approaches combine system modelling and domain-based visualization:

– *State-machines based* approaches, such as in Statecharts or UML state diagrams. Tools like Statemate and Rhapsody allow those FSM to be executed and coupled with graphical views. Recent work in the area explores reactive animations [5] in order to combine and merge efforts in reactive design (using statecharts) and animation techniques (such as Flash). Our approach has a similar purpose but focuses more on earlier steps: our FSM are less structured but are consistent with goals. Both approaches and tools look essentially complementary and definitively worth combining.

- *Tabular-based* approaches, such as SCR and RSML and supported with tools like SCR-toolset [7] and SpecTRM [9]. They focus on the system and model the input-output relation with a number of tables. The SCR animator has a visual composer with a library of display and control widgets which can be mapped on table variables. However the SCR library lacks a conceptual structure.
- *Labeled transition system (LTS)*. LTSA, an LTS animator relies on scenebeans, a descriptive XML-based animation framework [11]. It provides low level animation primitives (similar to SVG or Flash) which are easy to deploy and reuse even with other underlying modelling languages.
- In existing MB-IDEs approaches, the model that is usually chosen for representing the dialog, such as a state-transition diagram, a statechart or a Petri net, should be added and constructed by exploiting merely the task model, which leaves the problem typically underspecified. It is therefore the responsibility of the designer to manually choose the right design options and to tweak the dialog model to obtain a version that is detailed enough to work.

The above approaches are based on operational specifications and lack the ability to capture the goals, model the environment or reason about agents. Another goal-oriented approach is Formal Tropos which also has an animator but without visualization capabilities [6].

Finally note the focus in this work is on the the design of display/control panels for control systems—other approaches attempt to be more generic.

7 Conclusion and Future Work

In this paper, we exposed how goal models, borrowed from requirements engineering, can help in the design of control panels by providing a property driven paradigm as alternative to traditional data-oriented or task-oriented paradigms. A method to design consistent control panel user interfaces from goal models was proposed as well as a goal-oriented widget library to assemble them. The result has then been deployed on a requirements animator as a tool for designing agent-oriented animations. The work has been tested on a number of case studies such as a baggage lockers and a safety injection system. It is now being validated on a larger industrial case: a floor door system for an automated train.

FAUST and its animator mainly differ from existing MB-IDEs in that they exploit models borrowed from Requirements Engineering to produce a fully executable UI. In this way, we can also say that the underlying models are executable and dynamic, but they are also more expressive to the point that the dialog and a complete animation can be automatically generated. In addition, it is possible in existing MB-IDEs to apply model-checking techniques to verify properties of interest, such as IFIP Dialog principles, but they need to be run by a human operator. In contrast, here, there is no direct need to rely of these techniques because the resulting interface satisfies the FR and NFR by construction. It is impossible to produce a UI that is not compliant with the constraints imposed by the various models. In the case the requirements were somehow flawed/incorrect in the first place, the goal-directed methodology provides means to detect that because a conflict or inconsistency is likely to become apparent as the

refinement/agent assignment/operationalization process would fail at some point. Finally, the most interesting difference is that the first model, i.e. the goal model, contains both FR and NFR intertwined, as opposed to user goals only in traditional task-based models.

In the end, both approaches should not be presented as alternatives but rather as complementary: the focus is here more upstream in the design process and based on property refinement rather than task refinement. The resulting operation model can be seen as a task model which can be handled by standard HCI techniques. The benefit of using the presented RE-based approach is the access to interesting reasoning techniques about the system, its environment and the interactions across their borderlines which impact the UI. This calls for a more in-depth comparison of HCI and RE approaches and the ways they can cross-fertilize.

At short term, we envision to address a number of limitations. Currently, our work focuses mainly on control systems with agents responsible for a small part of the system. We would like to support supervising agents which have global monitoring/control capabilities on the system (eg., train dispatcher). Better structuring mechanisms are also needed for mapping more elaborated data type and helping in the reuse of preassembled sub-panels. We also plan to provide an integration of the goal monitor to reports alarms at interface level.

Acknowledgement

This work is financially supported by the European Union (ERDF and ESF) and the Walloon Region (DGTRE).

References

1. M. Biere, Birgit Bomsdorf, and Gerd Szwillus. The visual task model builder. In *Chapter 20, in Proceedings of CADUI'99*.
2. G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, L. Bouillon, and J. Vanderdonckt. A unifying reference framework for multi-target user interfaces. *Interacting with Computers*, 15(3), June 2003.
3. A. Dardenne, A. van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1-2):3–50, 1993.
4. R. Darimont and A. van Lamsweerde. Formal refinement patterns for goal-driven requirements elaboration. In *4th FSE ACM Symposium, San Francisco*, 1996.
5. S. Efroni, D. Harel, and I. Cohen. Reactive animation. In *Proc. 1st Int. Symposium on Formal Methods for Components and Objects (LNCS 2852)*, 2002.
6. A. Fuxman, L. Liu, M. Pistore, M. Roveri, and P. Traverso. Specifying and analysing early requirements in tropos. *Requirements Engineering Journal*, 2004.
7. C. Heitmeyer, J. Kirby, and B. Labaw. Tools for formal specification, verification, and validation of requirements. In *Proc. COMPASS '97, Gaithersburg*, 1997.
8. R. Koymans. *Specifying message passing and time-critical systems with temporal logic, LNCS 651*. Springer-Verlag, 1992.
9. G. Lee, J. Howard, and P. Anderson. Safety-critical requirements specification and analysis using spectrm. In *Proc. 2nd Meeting of the US Soft. Syst. Safety WG*, 2002.

10. E. Letier and A. van Lamsweerde. Agent-based tactics for goal-oriented requirements elaboration, 2002.
11. J. Magee, N. Pryce, D. Giannakopoulou, and J. Kramer. Graphical animation of behavior models. In *Proc. ICSE'2000, Limerick*, 2000.
12. Z. Manna and A. Pnueli. *The Reactive Behavior of Reactive and Concurrent System*. Springer-Verlag, 1992.
13. G. Mori, F. Paternó, and C. Santoro. Design and development of multi-device user interfaces through multiple logical descriptions. *IEEE TSE*, 30(8), August 2004.
14. F. Paternó. Model-based design and evaluation of interactive applications. In *Springer Verlag*, November 1999.
15. The Objectiver RE platform. http://www.objectiver.com.
16. C. Ponsard, P. Massonet, A. Rifaut, J.F. Molderez, A. van Lamsweerde, and H. Tran Van. Early verification and validation of mission critical systems. In *8th FMICS Workshop, Linz*, 2004.
17. A.R. Puerta. A model-based interface development environment. *IEEE Software*, 14(4), July/August 1997.
18. The FAUST toolbox. http://faust.cetic.be, 2004.
19. H. Tran Van, A. van Lamsweerde, P. Massonet, and C. Ponsard. Goal-oriented requirements animation. In *12th IEEE Int.Req.Eng.Conf., Kyoto*, September 2004.
20. A. van Lamsweerde and E. Letier. Handling obstacles in goal-oriented requirements engineering. *IEEE Transactions on Software Engineering, Special Issue on Exception Handling*, 26(10), October 2000.
21. J. Vanderdonckt. Advice-giving systems for selecting interaction objects. In *Proc. of 1st Int. Workshop on UI to Data Intensive Systems, Edimburgh*, 1999.