Executable Task Models

Tobias Klug
t.lastname@sap.com

SAP AG
SAP Research CEC Darmstadt
Bleichstraße 8, 64283 Darmstadt

Jussi Kangasharju
firstname@tk.informatik.tu-darmstadt.de

Department of Computer Science
Darmstadt University of Technology
Hochschulstr. 10, 64289 Darmstadt

ABSTRACT
Current task modeling techniques have a shortcoming in that they only use the model at design time. This means that the information contained in the model has to be embedded into the application and makes the task model static. In this paper we propose using the task model *at runtime*, in order to simplify producing applications which adapt to the actions of the user. In particular, we extend the ConcurTaskTree (CTT) notation to allow dynamic execution of a task model. A key feature of our extension is that it gives semantics for the use of information exchange operators. Our second contribution is an implementation of our dynamic task model. We present a prototype application which shows how the interactions with the task model at runtime allow us to produce a dynamic and context aware user interface.

Author Keywords
task modeling, dynamic UI construction, proactive user interfaces

ACM Classification Keywords
H5.2 User Interfaces – Graphical User Interfaces (GUI), D2.2 User interfaces – Dynamic Construction

INTRODUCTION
The user interface is one of the most critical factors for the success of a software product. If the users are not satisfied, the software risks being rejected. One of the main focus points in HCI research is to find ways of easing the development of easy-to-use user interfaces. One key step towards this goal is understanding the structure of the tasks that the user needs to do while using the application.

Task modeling is the art of capturing the tasks and their temporal relationships and dependencies on one another. Task models can be used at different stages of the development process, in analysis, design, and implementation phases. In the analysis phase, they can be used as a communication tool between developers and users that allows both parties to express their understanding of the future work process [2]. During the design and implementation phases, task models can be used as a starting point for the development of concrete user interfaces. Task models can also be used for quality assurance, because the application can be checked against the underlying model.

So far most approaches for using task models in the construction of user interfaces have followed a transformation-based approach. In this approach, the task model is transformed into more concrete representations of the user interface until an implementation for a specific platform, environment, or user is reached. At this point, however, all information about the original task model is lost. This has two consequences: First, all the information contained within the task model has to be compiled into the user interface. This can be very complicated. Second, a lot of flexibility is lost, because it is no longer possible to make changes to the task model at runtime to adapt the application to specific circumstances.

In highly dynamic environments, such as mobile and ubiquitous computing scenarios, these points become severe disadvantages. Mobile users tend to perform several tasks at the same time, interrupting and resuming a task repeatedly, leading to very complex temporal relations.

We believe that the correct solution for these problems is to use a dynamic version of the original task model which can be executed at runtime. This approach has several advantages. First, it allows applications to stay coherent with the workflow by accessing the runtime task model. Second, the task model can be modified on demand by performing the necessary changes to the task model in memory. Neither of these is possible with the traditional, statically compiled-in task models.

In this paper we present a system which is able to load a task model at runtime and allows an application to access and manipulate this task model. We present our extensions to the existing CTT notation and describe the semantics of our extensions. We also present an implementation of our executable task model and show one primitive and exemplary way of using it in a hospital ward round scenario.

This paper is structured as follows. First we review some related work in the area of task modeling. Then we present our

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
TAMODIA'05, September 26–27, 2005, Gdansk, Poland.
Copyright 2005 ACM 1-59593-220-8/00/0000…$5.00.

extension to the CTT notation. Finally, we present our implementation and the example application. In the conclusion we also present directions for future work.

RELATED WORK

The basis for our work is the ConcurTaskTree (CTT) notation developed by Mori et al. [5]. This notation enjoys widespread use throughout the research community for representing task models.

Several transformation based user interface creation tools have been developed. One example of a transformation-based user interface generation tool is the TERESIA environment [6]. It allows the interactive adaptation of the task model according to certain constraints from the platform as well as a semi-automatic transformation into an abstract user interface description (UIML [1]) which is further transformed into a concrete user interface.

Another example IKnowU [3] from Furtado et al. IKnowU is a knowledge based system that generates concrete user interfaces by applying design rules to abstract models. Those models and intermediate results are stored in UsiXML [4], an user interface markup language (UIML).

EXECUTABLE TASK MODEL

Task States

The CTT Environment already includes a task model simulator, which can be used for interactive walkthroughs of the developed task model. The purpose of the task simulator is to verify the task model during the design phase. One limiting assumption, that the simulator makes is that all leaf tasks are atomic. This has the following consequences. First, the simulator calculates a set of enabled tasks which represents the tasks that could be done next. Once one of these tasks is selected by the user for execution, it is automatically completed from start to finish. In other words, there is no intermediate state between the enabled and finished states. It is therefore impossible for two leaf tasks to be active at the same time, which contradicts the concurrent specification of the model. To correctly represent real world tasks, the use of the active state needs to be extended to the leaf tasks as well.

Task states and the transitions between that states are the main source of information when trying to capture current state of a task model. Figure 1 shows the complete set of states including the respective transitions as they are used in CTT. We believe that task states and transitions between those states, together with additional information about the nature of tasks, carry enough information to allow a basic understanding of the users tasks by an application accessing the model.

Information Exchange

The CTT notation also defines two information exchange operators, which are special versions of the enabling and the concurrency operators. These operators indicate, that the respective tasks are exchanging information. However, CTT offers no constructs to define what information is provided or consumed by specific tasks.

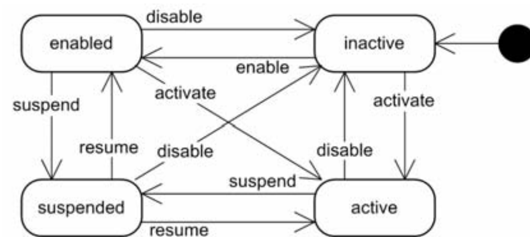

Figure 1. Task states and transitions in CTT.

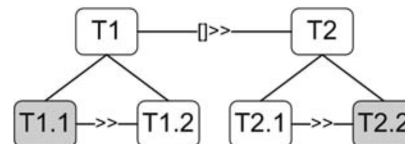

Figure 2. Task T1.1 produces information that is consumed by T2.2.

Our extension is to use the concept of input and output ports for each task. These ports can be seen as an interface of that particular task. We also need to define the semantics for the information exchange as well as rules to check the model for completeness and correctness.

Information exchange can happen between directly related tasks, but also between disconnected tasks. Consider the scenario in figure 2. Tasks T1.1 needs to send information to T2.2, but the two tasks are not directly connected. The standard information exchange operator of CTT can only be used between the high level tasks T1 and T2. Thus, in the CTT model, T1.1 cannot exchange information with T2.2.

Our input and output ports extend the information exchange possibilities and allow the communication between T1.1 and T2.2 to take place.

Input Port Semantics

Input ports indicate that a task or *one of its subtasks* is able to consume a specific piece of information. An input port is defined within the task that actually consumes the information. However, since the information exchange might happen at another (higher) level in the model, input ports are *inherited by all ancestors* of the task that has the input port. In our example, an input port A defined for T2.2 would be inherited by T2 because a subtask of T2 consumes this piece of information.

At runtime, tasks that receive information on an input port need to make sure that it reaches all descendants that can consume the information. Therefore information arriving on port A of T2 would also be transmitted to port A on T2.2. In a more complex hierarchy, all intermediate tasks would have to keep track of the input ports of their descendants in the task model.

Output Port Semantics

An output port indicates, that the task can provide some piece of information. Output ports are inherited in the same way as input ports. For example, if a port is defined on T1.1, T1 inherits that output port.

At runtime, output ports are stateful variables which carry the last value assigned to that port. If an output port is set on a task, this information is also transmitted to all parent tasks, making sure that all ancestors are able to provide the information. As a consequence, if several subtasks of a composite task provide the same information (i.e., have the same output port), only the last information provided any of the subtasks is stored.

How the information is transferred to related input ports depends on the operator being used:

- *A [ ] >> B (enabling with info exchange)*: When A finishes, the values of its output ports are transmitted to the respective input ports of B. Output ports without a matching input port are ignored.
- *A |[ ]| B (concurrency with info exchange)*: Whenever a new value is stored for an output port in A, that value is immediately transmitted to B. The same is also true if an output port changes in B, since this operator works in both directions.

Correctness and Completeness

A CTT model is only complete and correct if all tasks with input ports are guaranteed to receive the desired information through a corresponding output port on a related task. But as tasks can be skipped during execution, special care has to be taken. For a composite task B, the model checker needs to make sure, that every possible activation path through the subtasks of B includes at least one task with the appropriate output port.

IMPLEMENTATION

We have implemented a prototype executable task model in Java. Our goal in the implementation was to separate the application logic from the underlying model semantics using a model/view/controller-like pattern. Therefore, each task is separated into two objects. A `BasicTask` node handles the CTT model semantics (the model) and an object implementing the `TaskFunctionality` interface performs the actual work of the tasks (controller).

The `BasicTask` nodes contain all information necessary to execute the task model correctly. This includes the task's name, its temporal operator, the current state, as well as the input and output ports. Changes to the model are communicated to the respective `TaskFunctionality` through events.

Classes implementing the `TaskFunctionality` interface handle events thrown by the underlying `BasicTask`. These events are: *task enabled*, *task activated*, *task finished*, *task suspended*, *task resumed*, *task disabled*, *new value on input port* and *new value on output port*.

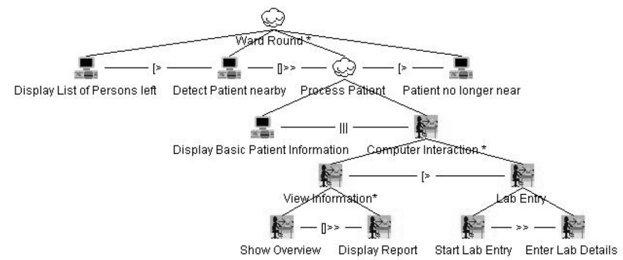

Figure 3. Simplified ward round task model.

A single application therefore consists of several components: a task model that is loaded from an XML file, application specific `TaskFunctionality`'s, a mapping between tasks and specific implementations. Optionally the application might require other components like backend systems and UI components.

APPLICATION

An executable task model does not create intelligent applications by itself. The application still needs to interpret and use the information conveyed by the task model. We used our executable task model architecture to develop a simple prototype for a system accessing a task model updated at runtime. The prototype semi-automatically generates a user interface from an underlying task model using pre-programmed UI blocks. The scenario we use is a medical ward round scenario with context awareness.

Scenario description: *When the doctor approaches a patient, the system detects the patient and automatically displays relevant information. The doctor can then choose to view specific reports or order examinations and lab analyses. As soon as the patient is no longer nearby, the system aborts all current activities and goes back to the initial state (see figure 3).*

There are two particular challenges in this scenario. First, the workflow is heavily customized to a specific doctor and therefore it is necessary that the application can be easily reconfigured for different ward round styles. Second, the application also has to be very flexible during the ward round because doctors possibly perform several tasks at the same time.

User Interface

The user interface is constructed dynamically from the current state of the task model. The application recognizes three different cases:

- *Active tasks*: The user interface is implemented as a stack of pre-defined GUI panels. Each GUI task is represented by one panel. As long as a task is active, this panel is dynamically added to the GUI stack (figure 4).
- *Enabled interaction tasks*: Tasks in this state are rendered as a button on the bottom of the stack. By clicking a specific button, the user indicates that he wants to activate that particular task (figure 4).

Figure 4. Automatically generated GUI stack before (top) and after activating the task “Start Lab Entry” (bottom).

- *Enabled application tasks*: These tasks require no user interface since the application has full control. One example is the patient detection task that is activated and immediately finished as soon as a patient is detected.

The application task “Detect Patient Nearby” demonstrates the use of information exchange. An identification number of the detected patient is transmitted to various tasks belonging to “Process Patient”. These use the information to display patient-specific data (figure 4).

The application is also easily reconfigurable with regards to the temporal structure. If another doctor does not want the report he is reading to disappear when he starts lab entry, the operator between “View Information” and “Lab Entry” can be changed from disabling to concurrency.

CONCLUSION

In this paper we have extended the CTT notation and semantics to allow us to keep an instance of the task model in memory, such that we can update the model in real-time. Our extensions include the active task state for leaf tasks which allows them to be non-atomic. Also we have extended the information exchange operators by defining input and output ports, as well as rules to check for model consistency and completeness.

We have further shown how an application can utilize such an executable task model to dynamically assemble a user interface from available blocks, according to the status of the task model. Our example automatically inherits the correct temporal behavior according to the underlying model.

The example application used in this work is simple and more complex schemes for the assembly of the UI can be envisioned. We are currently investigating schemes that do not rely on the availability of pre-programmed UI blocks.

Our research on using task models at runtime is still in progress. The executable model is meant as a prototype that further research can be based upon. We plan to increase the amount of information associated with each task and will investigate how this information can be used effectively to enhance dynamic applications.

REFERENCES

1. Abrams, M., Phanouriou, C., Batongbacal, A., Williams, S., Shuster, J. *UIML: An Appliance-Independent XML User Interface Language*, In *Proceedings of the 8th WWW conference*, 1999.
2. Barbosa, S.D.J., Greco de Paula, M., Pereira de Lucena, C.J., *Adopting a Communication-Centered Design Approach to Support Interdisciplinary Design Teams*. In *SE-HCI workshop at ICSE 2004*, Edinburgh
3. Furtado, E., Furtado, V., Sousa K.S., Vanderdonckt, J., Limbourg, Q. *KnowiXML: A Knowledge/Based System Generating Multiple Abstract User Interfaces in USIXML*. In *Proc. TAMODIA 2004*, Prague, 121–128
4. Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., Florins, M., Trevisan, D., *UsiXML: A User Interface Description Language for Context-Sensitive User Interfaces*, In *Proc. of the ACM AVI’2004 Workshop*
5. Mori, G., Paternò, F., Santoro, C. *CTTE: Support for Developing and Analyzing Task Models for Interactive System Design*. In *IEEE Transactions on Software Engineering*, 2002, Vol. 28, No. 9
6. Mori, G., Paternò, F., Santoro, S. *Tool Support for Designing Nomadic Applications*. In *Proc. of IUI 2003*, Miami, 141–148