Methods for efficient development of task-based applications

Vaclav Slovacek[1],

[1] Dept. of Computer Graphics and Interaction, Faculty of Electrical Engineering, Czech Technical University in Prague, Karlovo nám. 13, 121 35, Praha 2, Czech Republic
slovavac@fel.cvut.cz

Abstract. This paper introduces methods for developing task-based applications by tightly integrating workflows with application logic written in an imperative programming language and automatically completing workflows especially with tasks that mediate interaction with users. Developers are then provided with completed workflow they may be used for further development. Automatic completion of workflows should enable to significantly shorten the development process and eliminate repetitive and error-prone development tasks. Information extracted from workflow structure and low level application logic may then be used to automatically generate low to high fidelity prototype user interfaces for different devices and contexts.

Keywords: Workflow, workflow processing, task modeling, generated user interface

1 Introduction

The main goal of the research related to this paper is to introduce methodology that would enable efficient development of task-based applications using visual task modeling and present functionality of an ongoing task framework implementation.

Developing applications by first designing a task model and then automatically generate user interface provides developers with option to model application on higher levels of abstraction as task models abstract from device display resolutions, input methods, available user interface components, etc.

Modeling an application using workflows provides formal description of all processes in the application that is understandable by non-programmers. It enables to design and analyze the application on different levels of abstraction (using nested workflows) and enables eventually to detect design issues in the processes design [1].

Despite all the advantages listed above, current methods also suffer from significant drawbacks that prevent many developers from adopting task modeling as a method for developing applications. Developers have to learn formal semantics and although workflow schemas are easy to read they are much harder to design properly, especially when different tasks may run in parallel, cancel each other, etc.

Workflows may be used to automatically generate user interfaces (for example web forms that asks user for required input) and we believe that by making designing workflows simpler the workflows may be then used at least for rapidly designing low

fidelity prototypes that may be modified by user interface designers, used for usability testing and finally converted to final user interfaces accelerating development cycle.

2 State of the art

Common tool for task modeling are ConcurTaskTrees [2] that are used for hierarchical task analysis [3]. ConcurTaskTrees describe a high level task by hierarchically splitting it into subtasks. Abstract tasks that represent users intentions are split into elementary tasks. These tasks are either machine tasks (performed by a device a user is interacting with), user tasks (performed by a user) and interaction tasks (user interacting with the device). Complex branched processes are easier to express using workflow languages such as YAWL [4] and BPEL [5] that enable do define branching, iterations, etc. There are also several implementations of a workflow engines and visual editors available for both YAWL and BPEL.

There are several projects focused on automatic user interface generation. Project SUPPLE[6] currently enables to automatically generate user interfaces. It is possible to provide SUPPLE with data that are required to be entered by a user (specifying the type of data, name, allowed and expected values) and SUPPLE provides a user interface optimized for a specific user [7] enabling the user to enter the required data.

3 Framework

We focused on making the process of designing task-based applications as usable for developers as possible. We took advantage of currently existing technologies and built on knowledge of developer enabling them to quickly start developing more maintainable applications while providing methods for rapidly deliver low to high fidelity user interface prototypes that are integrated with application logic.

3.1 Workflow

We have chosen YAWL as a base language for our research because it covers all necessary patterns for describing any application logic unlike BPEL [4]. Also every BPEL process can be converted to an appropriate YAWL workflow [8]. We use a simplified YAWL notation closely described in [9] that reduces number of visual elements to describe a workflow.

We also do not use XPath and XQuery that are used in YAWL for branching conditions and data updates. Instead we delegate branching logic to an imperative programming language such as Java and provide an API for controlling execution of the workflow (e.g., choosing tasks should be executed). Using an imperative programming language for elementary application logic should be natural for most developers.

3.2 Task types

Workflow is a descriptive form of defining tasks. Although workflows are efficient for describing high level processes, it generally fails to simply describe low-level application logic [10] that is much more efficiently described using imperative programming languages such as Java, ECMAScript, etc. By low-level application logic we understand operations that have no inner structure that should be exposed.

The problem with using an imperative language for implementing elementary tasks is that it is very hard to limit their functionality so it does not perform operations that should be rather defined in a workflow. Decision what is still considered elementary task depends on developers and thus a set of rules and recommendations for proper coding style should be introduced. Otherwise advantages of having a descriptive task-model may be lost as most of the application logic might be implemented in elementary tasks that have no inner structure and behave as black boxes.

We split tasks into implicit, triggered and user task types. These types differ by when they are executed. Implicit tasks are executed by the workflow engine automatically when they are reached in an executed workflow. Triggered tasks must be initialized from a user interface and are easily recognizable as they require an input data. Server stops processing workflow until it receives a triggering event with the required data from a user interface.

User tasks are similar to triggered tasks, but are directly accessible by user, so there must be a button or another interaction element visible that enables users to initialize the task. Developer is responsible for declaring user tasks to distinguish them from triggered tasks (e.g., using *@UserTask* annotation). This information is important for automatically generating user interfaces.

3.3 Execution conditions

Execution conditions are conditions that must be satisfied for a task to be available for user. The conditions in our framework are identified by a unique id (e.g., fully qualified name of class that is used to evaluate if the condition is satisfied).

As the framework abstracts from how execution conditions are represented. It is possible to extend the support of execution conditions to Java Bean Validation [11], semantically described conditions, etc.

3.4 Execution condition satisfiers

Execution condition satisfier is a task that is automatically inserted before another task to satisfy its execution conditions. The condition satisfier itself may require different execution conditions to be satisfied and thus it might be necessary to add another condition satisfier preceding it (e.g., condition satisfier producing a user object instance, may require user name and password).

The framework abstracts from how a condition satisfier task satisfies an execution condition so different implementations may be used - e.g., implementation providing manually designed user interface to a user, implementation providing automatically generated user interfaces, implementation providing static data, etc.

4 Modifying workflow

To simplify and accelerate workflow-based application development we propose a method for modifying workflow that completes the workflow automatically based on execution conditions extracted from source code of elementary tasks.

An application is a set of tasks the user may perform (the tasks the application was designed to perform). These tasks are usually implemented in elementary tasks and the rest of the workflow ensures they are performed in the desired order, under certain conditions (e.g., time, location), after required data get available, etc.

4.1 Satisfying execution conditions

The figure 1 shows four different representations of the same workflow. The workflow as defined by developer (a) contains just one task that requires a condition uniquely identified by the letter A (shown in the circle on the left side of the task) to be satisfied.

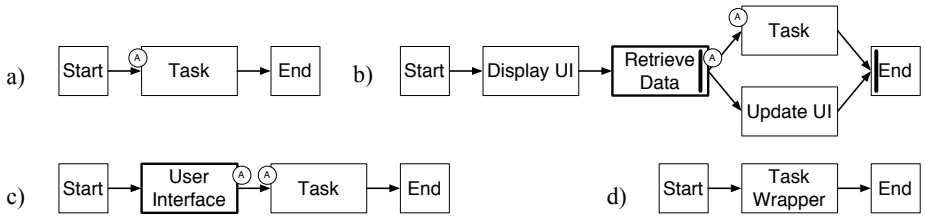

Fig. 1. The same task represented in 4 different ways - a) as defined by a developer, b) automatically completed, c) simplified semantic representation that is more readable by a developer, but not is semantically equivalent with (b), d) wrapping task with complex inner structure in form of nested workflow

The framework modifies the workflow (b) by adding a task (Display UI) that renders a user interface visible to user and then the execution stalls before executing task Retrieve Data waiting for user input (tasks requiring input from a user interface are marked by a thicker border). After a user provides the required data the original task is executed in parallel with another added tasks that notifies user about progress of the original task which may be important for user experience [12] (e.g., displaying a progress bar, showing notification when the task is finished, etc., depending on implementation).

Although the automatically completed workflow exactly represents the application behavior it is quite complex and adding such constructs to every task with unsatisfied execution conditions would lead to overly complex structure that might be difficult to work with.

Because of this the framework might represent the task by merging all the added tasks into a single task (User Interface) that precedes the original task as shown in (c), this however leads to losing the parallel branch that lets user informed about progress of the task.

The only acceptable pattern seems to be creating a wrapping task that encapsulates a nested workflow similar to the one shown in (b). As all execution conditions are

solved inside the nested workflow there is no unsatisfied execution condition on the higher level and the higher level workflow is kept simple compared to (b) and (c).

4.2 Completing workflow

Satisfying an execution condition a single task is useful for dealing with very simple patterns. Eventually satisfying execution conditions for each task individually may lead to generating poor user interfaces.

The figure 2 shows an example workflow having two sequentially executed tasks (a). Using the method described in the previous section the framework would provide two separate user interfaces preceding each task. However there is no other branching or conditional logic that would cancel the sequential execution thus the execution conditions may be collected and a user interface preceding execution of the first task may be provided asking user for input required for both tasks as shown in (b).

A different pattern is shown in (c) when a common user interface is used to get input necessary for both task and then asks for additional input necessary only for one of the branches (d) while the other branch may be executed while waiting for a user to input necessary data.

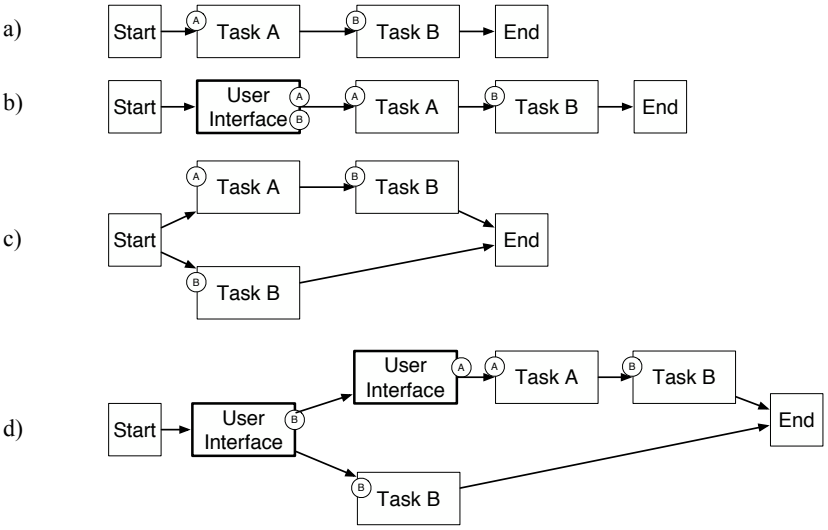

Fig. 2. Example workflows as defined by a developer (a), (c) and with automatically added condition satisfier tasks (b), (d)

5 Metadata related to user interfaces

Information necessary for rendering user interfaces is directly extracted from elementary task source code. A lot of information may be extracted from the source code using language reflection, which is supported by several programming

languages[13], without a developer having to provide any additional information. The following Java code is an example of an elementary task as used in our framework:

```
@Title("title") @Description("description")
class Login implements UserInteraction {
   void execute(
      @NotNull DeviceContext requiredDeviceContext,
      @Title("title") @Description("description")
      @Input("username") String name,
      @Input("password") String password) {
         // application logic here
   }
}
```

The above code snippet contains enough information to be properly represented in a user interface. The framework extracts fields of the class and parameters of the *execute()* method. Based on their types and names defined in *@Input* annotation it injects appropriate values to class fields using inversion of control mechanism before calling the *execute()* method with appropriate parameters.

The class implements interface *UserInteraction* that extends interface *Interaction* common for every elementary task. As interfaces support inheritance they may be used to organize interactions into different groups. This may be used for better layout of automatically generated user interface (e.g., clustering of related user interface elements).

Additionally the code snippet above contains annotations (*@Title* and *@Description*) that provide closer description of both the interaction itself and the parameter expected to be provided by a user. These information may be used to properly represent the task in a user interface. We have introduced annotations that are a copy of Dublin Core [14] metadata tags used typically in XML to describe these properties of elementary tasks. The title may be for example used for a label related to a text field for input of the appropriate data and description for providing a tooltip closer describing the required input.

5.1 Task importance

There are several ways how importance of tasks for a given application may be calculated. The framework enables to specify the importance of certain tasks manually (e.g., using *@Importance* annotation). This is a mechanism that tells which tasks are necessary in the application. This typically includes the tasks the application is designed for (e.g., volume control and channel switching in a TV application).

However specifying importance manually to all tasks is not very useful. Thus importance may also depend on user preferences (for example unexperienced user will not access color management settings of a TV), on an end-device used or any other environment properties.

Calculating transitive importance is used to propagate importance of individual tasks forward in the workflow. We simply do that by propagating the highest importance to previous tasks in a sequence. This ensures that the task is properly represented in the interface that is provided to user before it is even reached in a

workflow. An example of the importance propagation is shown in figure 3 with workflow (b) showing transitive importance calculated from workflow (a) and workflow (d) showing transitive importance from workflow (c). In workflow (d) the importance is propagated only to a single branch that already has higher importance to prevent cluttering a user interface.

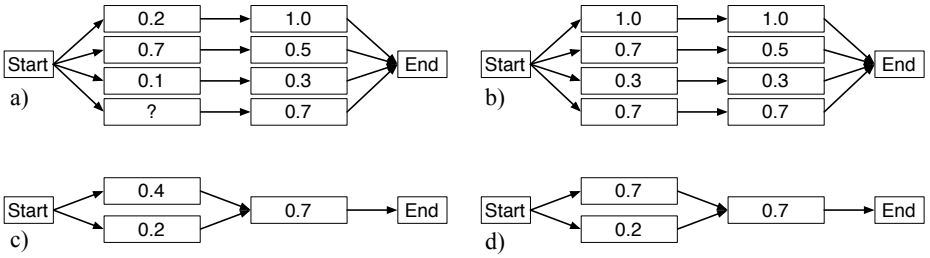

Fig. 3. Workflow with tasks with calculated importance (a) parallel tasks with individually calculated importance, (b) the workflow with transitive calculated importance, (c) sequential tasks with individually calculated importance, (d) the same workflow with transitive calculated importance

Calculated importance may then be used also for automatically generating user interfaces where more important tasks may be represented e.g., by larger buttons, more prominently placed controls, etc. Tasks with importance under a defined threshold may eventually not be visible in a very constrained environment (e.g., very small device screen).

We distinguish three different states of a workflow task. Task may be enabled (all execution conditions are satisfied) meaning that user may execute it. Task may disabled (at least one execution condition is not satisfied) causing all relevant user interface elements (e.g., button executing the task) to be disabled but still visible in the user interface thus keeping it consistent. Tasks that are in hidden state are not represented in the user interface and their controls simply disappear. As it would be difficult for a computer system to guess whether the tasks should be disabled or hidden developer has to declare that a task switches to disabled state (e.g., by adding *@AlwaysVisible* annotation).

6 Conclusion

We have introduced basic methods for automatically completing workflow models for developing applications that should simplify development of applications based on workflows.

We have strongly focused on process that would enable to automatically generate user interfaces directly from application logic implemented in an imperative programming language while managing the state transitions based on workflow description tightly integrated with an application low-level source code.

7 Future research

We have not covered a situation that may occur when an execution condition is first satisfied by condition satisfier task, then concurrently made unsatisfied by another tasks and then a tasks requiring the execution condition to be satisfied is reached.

It would be possible to enclose blocks that depend on satisfying execution conditions into transactions. However as these transactions might involve user interaction there may arise problems with very long transactions blocking other processes in an application and problems with selecting proper items in a context to be locked. Transactions locking large part of a context for a long time may result in deadlock and/or may significantly slow down an application.

Research should also focus on usability of automatically generated user interfaces and analyze impact of methods described in this paper on development process.

References

1. Van Breugel, F. and Koshkina, M. Models and Verification of BPEL. *Unpublished Draft*(Jan 1 2006).
2. Paternò, F., Mancini, C. and Meniconi, S. *ConcurTaskTrees: A diagrammatic notation for specifying task models*. Chapman & Hall, Ltd. London, UK, UK, City, 1997.
3. Stanton, N. Hierarchical task analysis: Developments, applications, and extensions. *Applied Ergonomics*(Jan 1 2006).
4. Van der Aalst, W. and Ter Hofstede, A. YAWL: yet another workflow language. *Information Systems*, 30, 4 2005), 245-275.
5. Juric, M. B. Business Process Execution Language for Web Services BPEL and BPEL4WS 2nd Edition2006).
6. Gajos, K. and Weld, D. SUPPLE: automatically generating user interfaces. *Proceedings of the 9th international conference ...*(Jan 1 2004).
7. Gajos, K., Wobbrock, J. and Weld, D. Automatically generating user interfaces adapted to users' motor and vision …. ... *symposium on User interface ...*(Jan 1 2007).
8. Brogi, A. and Popescu, R. From BPEL processes to YAWL workflows. *Lecture notes in computer science*, 41842006), 107-122.
9. Slovacek, V. Towards Workflow-based Application Development Framework2010).
10. Jelinek, J. and Slavik, P. GUI generation from annotated source code. *Proceedings of the 3rd annual conference on ...*(Jan 1 2004).
11. Bernard, E. and Peterson, S. JSR-303 Bean Validation. *Bean Validation Expert Group*(Jan 1 2009).
12. Blackmon, M. Cognitive walkthrough. *Encyclopedia of human-computer interaction*.
13. Gosling, J., Joy, B., Steele, G. and Bracha, G. *Java (TM) Language Specification, The (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.
14. Weibel, S., Kunze, J., Lagoze, C. and Wolf, M. RFC2413: Dublin Core Metadata for Resource Discovery. *RFC Editor United States*(Jan 1 1998).