Transactions in Task Models

Daniel Reichart, Peter Forbrig

University of Rostock, Department of Computer Science
{daniel.reichart|peter.forbrig}@uni-rostock.de

Abstract. In this paper we propose a method to model the behaviour of task models in error situations. For these purposes we follow the idea of transactions in database systems. By encapsulating tasks in transactions the atomicity of complex tasks can be asserted. Corresponding tool support is presented which includes modelling and simulating task models. The tools themselves were developed in a model-based way.

Keywords. Transaction, Task Model, Tool Support

1 Motivation

The diversity of mobile devices and platforms requires new methods to master the complexity of user-interface development. Abstract models can help to solve many issues so that model-based user interface development becomes more and more popular. Task models are widely used to specify interactive software. Many methods and tools using task models to develop user interfaces. But still there are many problems that can occur, when generating user interfaces from these models. Task models just describe interactions between user and system in an idealistic way. Exceptions to this default behaviour is hard to express or even can not be expressed. But in real world applications errors occur and developers have to specify fallback behaviour. What happens, if a system task fails, because a required resource is not available? Which tasks have to be undone to get back to a consistent state? The cascading selective undo mechanism presented in [1] can help to address the second question but has another motivation. Instead of undoing selective, already successfully completed tasks and their impact on application state we propose an approach to handle error recovery strategies for task models using the concept of transactions.

2 Transactions

Transactions were originally developed to be used in database management systems to avoid inconsistencies of data. Such problems can arise when two processes

write the same data concurrently or in case of hardware or network failures. The idea of this paper is to encapsulate more than one task into one transaction. The three new operations *begin*, *commit* and *rollback* define the boundaries of the transaction. Transactions in databases are required to ensure the following constraints:

- Atomicity: Atomity guarantees, that either all of the operations are performed or none of them.
- Consistency: The database remains in a consistent state before the start and after the end of the transaction.
- Isolation: Isolation ensures, that each transaction appears to be isolated from all other transactions. This means, an operation outside a transaction can not see intermediate data of the transaction causing unwanted side effects.
- Durability: Durability guarantees, that once a transaction was performed successful it will persist.

These so called ACID criteria are too strict to be used in workflow systems or task models. To loosen some of the restrictions there are advanced transaction models to specify nested transactions [1], long-living transactions [3] or multi-level transactions [4]. We make use of some of these ideas and concepts in modeling transactions in task models.

3 Task models

The task models we are dealing with are derived from the CTT notation [5].

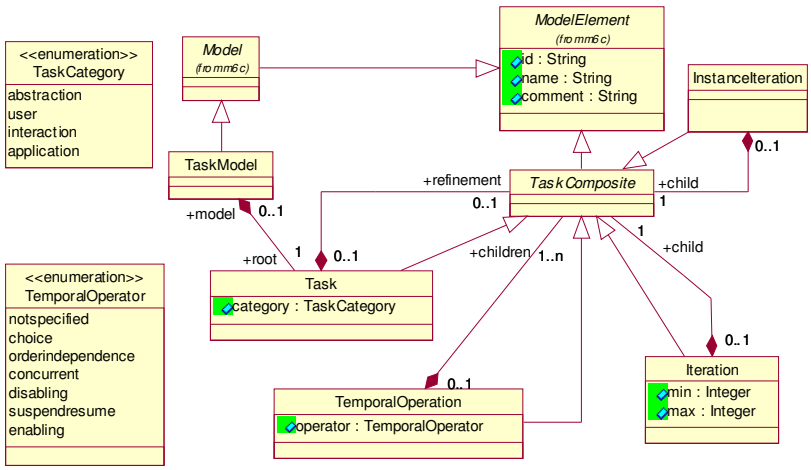

Fig. 1. task-meta-model

A task model is basically a tree of tasks and subtasks. Iterations and optional tasks can be specified as well as different temporal relations between subtasks.

Figure 1 shows the important parts of our task-meta-model. This meta model is an integral part of our tool development process [7, 8]. Using Eclipse [9] and some

frameworks like EMF [10], GEF [11] and GMF [12] we developed a set of model-based user interface design tools.

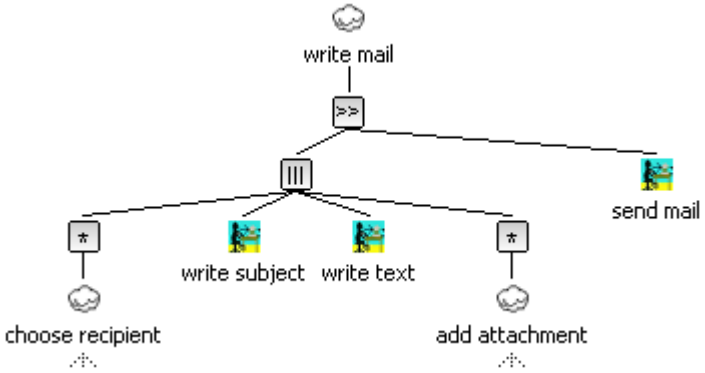

Fig. 2. task model "write mail"

Figure 2 shows an example task model created with one of our tools. It differs a little bit from the CTT notation. Temporal relations and iterations are nodes in our models instead of attributes respectively associations. One advantage of this notation, that one can immediately see the order of applied temporal operations without knowing operator priorities like in CTTE.

3.1 Lifecycle of tasks

Each task passes different states during its lifetime. A state chart can be used to specify the states and possible transitions between them, like in [13]. We developed our own state chart that fits our needs.

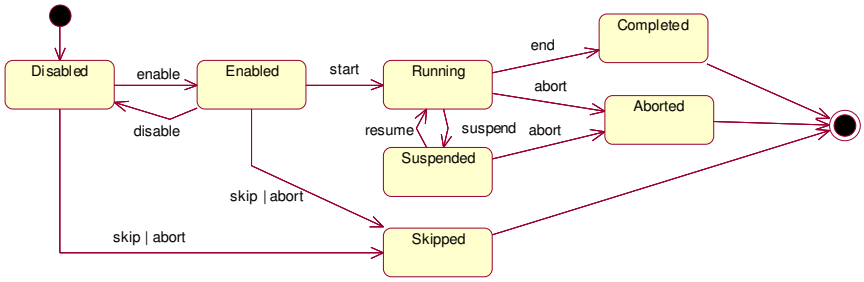

Fig. 3. lifecycle of a task

This state chart of Fig. 3 is applicable for basic (leaf) tasks as well as complex tasks. At the beginning, a task is in the state *Disabled*. In the default case, the event *enable* causes a state change to *Enabled*, *start* changes the state to *Running* and *end* results in the final state *Completed*. Variations of this behaviour arise by using different temporal operators. For example, using a *Choice* operator between two tasks A and B, *skip* is send to task A when the user chooses to start task B, effecting in state *Skipped*. The operator *OrderIndependence* takes care that while one task is running

the other task will be temporarely disabled by sending *disable*. The events *suspend* and *resume* occur using the temporal operator *Suspend/Resume* and *abort* is sent by the operator *Disabling* to cancel task A when task B starts.

To simulate a complete task model, for each task an instance is created first. This instance contains amongst other things the current state of execution, following the above state chart. The temporal operators act like agents between these instances and take care to reproduce the specified behaviour. For example, the temporal operator *Enabling* between two tasks A and B achieves this by observing the state of A and send the event *enable* to B when A changes his state to *Completed*.

3.2 Transactions in task models

The reason to introduce the concept of transactions into task models was to model the behaviour in case of an error. First, we had to reflect error situations in our runtime models. We inserted a new state *Failed* into the state chart and a transition from *Running* to *Failed*, reflecting an error situation. When a task enters the state *Failed*, interesting questions arise: What happens with the state of following tasks and the parent task? How can the task model get back to a consistent state?

We take a look at some examples first: Let's assume, in figure 2 the task *send mail* cannot be performed due to connection problems. The reasonable behaviour here is to give the user the opportunity to retry the task *send mail* when the network connection is working again.

In another task model we describe a complex calculation. If on of it steps cannot be performed, e.g. if some data is missing, the whole calculation fails due to missing intermediate data.

A third task model contains the task of booking a journey. This includes amongst other things the booking of a flight, a hotel and a rental car and the payment process. If one of these steps goes wrong (no hotel available, not enough money, …) any already performed task has to be undone. This behaviour is similar to the rollback operation of a transaction.

There may be other strategies to handle errors in task models but we will focus upon the three strategies described above: try again, abort and roll back. We extended our task models by adding an attribute for each task to specify, which strategy to apply.

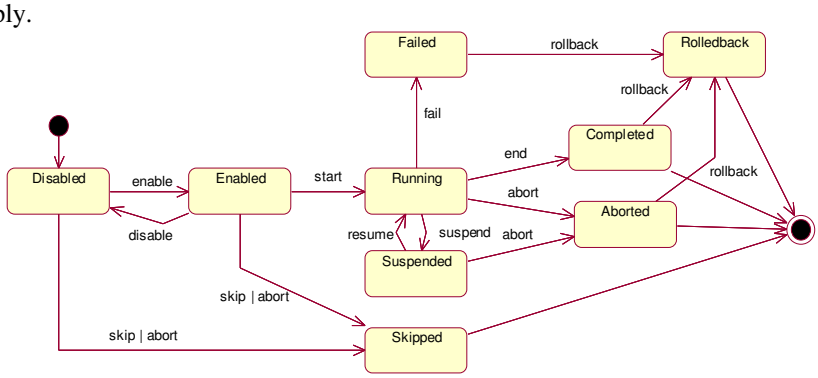

Fig. 4. Extended lifecycle with transaction concepts

Figure 4 shows the extended lifecycle of a task, including the two new states, *Failed* and *Rolledback*. We also defined for each combination of temporal operator and strategy, how to behave, when a tasks state switches into the state Failed.

The strategy "Abort" generally causes a failure of the task when a subtask fails. Using this strategy all over the task model, each failure in one of the subtasks causes the whole model to fail.

"Try again" resets the task and all of its subtasks when a subtask fails. Using this strategy we can stop the error propagation from a leaf task to the root task resulting from the application of the strategy "Abort".

The strategy "Roll back" revokes already performed tasks by executing the opposite tasks in reversed order, for example the cancelation of orders or accounting transactions. Using this strategy we create an effect similar to transactions in database systems: Either the whole tasks is performed or nothing. Of course, not all criteria of database transactions are fulfilled, but this is not required.

3.3 Tool Support for transactions in task models

To test the above ideas we implemented them in a few of our tools. First of all, we enhanced the meta model in figure 1 and added an attribute to specify for each task, which strategy to apply and how many times the user can retry a task. For example, the task model designer can specify, that the user has 3 attempts to perform "enter PIN", until this task fails finally. These meta-model-changes are reflected directly in our editors.

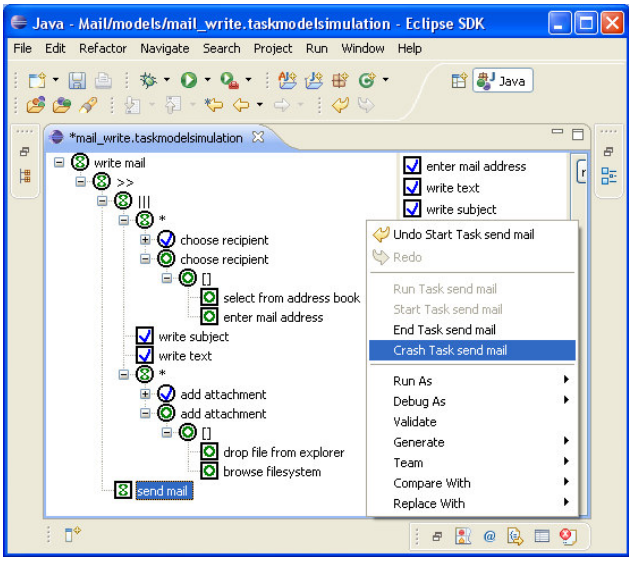

Fig. 5. Simulation of a task model

Further modifications are related to our task model simulation engine: The introduction of the new task states *Failed* and *Rolledback* and the implementation of

error strategies. The user interface to control the task model simulation has changed too: Users are able to send the message *Crash* to a task to simulate an error as seen in figure 6.

Additionally, the order of already performed tasks can be seen now on the right side to keep an eye on how the rollback mechanism works. In this example, the tasks *enter mail address*, *write text*, *write subject* and *drop file from explorer* (hidden by the popup menu) are already completed.

4 Summary and future work

The paper discussed an approach to address error situation in task models, using ideas from the concept of transactions. In the process of developing user interfaces we need to use this method to specify non-standard cases in task execution. This approach works on a very basal level. It does not consider consistency on the object level. For example, if a task modifies the state of an object and is rolled back later, the object's state will not be restored.

In the future we want to readjust our other tools, like the dialog graph editor [8] to the task model transaction approach. We have to develop new concepts for dialog graphs in order to react reasonable to error situations in task models.

References

1. Cass, A., Fernandes, C. : Using Task Models for Cascading Selective Undo, TAMODIA 2006, Hasselt, Belgium, 2006.
2. Moss, J.: Nested Transactions and Reliable Distributed Computing. In: Proc. Of the 2nd Symposium on Reliability in Distributed Software and Database Systems, 1982.
3. Garcia-Molina, H., Salem, K.: Sagas. In: Proc of ACM SIGMOD Conference on Management of Data, 1987.
4. Weikum, G., Schek, H.: Concepts and Applications of Multilevel Transactions and Open-nested Transactions. In: Database Transaction Models for Advanced Applications, 1992.
5. CTTE: The ConcurTaskTree Environment. http://giove.cnuce.cnr.it/ctte.html
6. Sinnig, D., Wurdel, M., Forbrig, P., Chalin, P., Khendek, F.: Practical Extensions for Task Models, TAMODIA 2007, Toulouse, France, 2007.
7. Brüning, J., Dittmar, A., Forbrig, P., Reichart, D.: Getting SW Engineers on Board: Task Modelling with Activity Diagrams, EIS 2007, Salamanca, Spain, 2007.
8. Forbrig, P., Reichart, D.: Ein Werkzeug zur Spezifikation von Dialoggraphen, Mensch und Computer 2007, Weimar, Germany, 2007.
9. Eclipse. http://www.eclipse.org (visited: 2008/06/08)
10. Eclipse Modeling Framework http://www.eclipse.org/emf (visited: 2008/06/08)
11. Graphical Editing Framework http://www.eclipse.org/gef (visited: 2008/06/08)
12. Graphical Modeling Framework http://www.eclipse.org/gmf (visited: 2008/06/08)
13. Bomsdorf, B.: The WebTaskModel Approach to Web Process Modelling, TAMODIA 2007, Toulouse, France, 2007.