Exception Handling Through a Workflow

Hernâni Mourão[1], and Pedro Antunes[2]

[1] Escola Superior de Ciências Empresariais, Instituto Politécnico de Setúbal, Campus do IPS – Estefanilha, 2914-503 Setúbal, Portugal, and
LASIGE (Laboratory of Large Scale Information Systems)
hmourao@esce.ips.pt
[2] Faculdade de Ciências, Universidade de Lisboa, Departamento de Informática, Campo Grande – Edifício C5, 1749-016 Lisboa, Portugal, and LASIGE (Laboratory of Large Scale Information Systems)
paa@di.fc.ul.pt

Abstract. Exception handling is a fundamental functionality of workflow management systems (WfMS). User involvement in exception handling is recognized as critical in various situations due to the unpredictability nature of the exceptions that can occur in a running workflow (WF) engine. The problem however is how to orchestrate human ad hoc interventions with a minimum impact on system integrity. The control flow and data integrity dimensions of the impact are analyzed. Our perspective is to allow the maximum latitude possible to user interventions while keeping system correctness. We propose a solution that uses a WF to guide users handling WF exceptions. Furthermore, we extended the WF engine with a propagation mechanism allowing users to involve multiple members of the organization in the exception handling WF. This solution is implemented in the OpenSymphony (OS) platform. The implementation details of the proposed solution in the OS platform are also given in the paper.

1. Introduction

Exception handling in WfMS is fundamental to react to situations that differ from the normal behaviour of the designed processes and is critical to successful implementation in real world scenarios [1; 12; 24].

There are two types of events that require non-standard WF behaviour [29]: 1) some specific requirements of an instance running on the WFMS requiring special attention (*ad hoc* changes); and 2) due to new legislation, strategy or reengineering efforts the company has to change the business model (*dynamic or evolutionary* change). In the former situation, changes have an impact at the instance level, while in the later situation a new model is defined for all instances of a specific class.

On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE: OTM Confederated International Conferences, CoopIS, DOA, and ODBASE, Lecture Notes in Computer Science, vol. 3290, pp. 37-54, 2004, R. Robert Meersman and Z. Tari, Eds. Heidelberg: Springer-Verlag.

Usually, the timing associated with dynamic changes allows proper planning [10]. This technique has been deeply studied [2; 4; 10; 15; 23; 25; 29].

Our work is focused on *ad hoc* interventions, where the change cannot be predicted in advance nor proper planning is usually feasible. In this type of situations the user involvement is carried out on a problem-solving basis [2; 6; 11; 14; 15; 31]. Moreover, a coordinated effort among all the persons involved in problem solving is crucial to overcome the situation.

The problem then is how to involve humans in exception identification and recovery while preserving the WF engine integrity. In this paper we developed an approach, introduced in [20], to support such human interventions. The basic solution consists in developing a toolkit of identification and recovery components. As a toolkit, this approach offers the flexibility, compositionality and extensibility necessary to allow humans handling exceptional situations. As a collection of individual components, each one must be developed to preserve the WF engine integrity. The toolkit exploitation is supported by a special WF dedicated to model and control the exception handling process (thus, exception handling is a process [10; 26]).

Two fundamental concerns have guided the implementation of this solution. One is that data describing the exceptional event is crucial to guide humans through the execution of recovery actions. The second issue is that an exception sometimes emerges as a series of events that travel throughout the organization, rather than one single exceptional event. As a consequence of these two concerns, the implemented solution also offers:

- A situation awareness component, gathering information about the exceptional events, implicated processes and engine status. This information may be gathered from the system (e.g. event types) and humans involved in the process (e.g. characterization of the exceptional event).
- An exception propagation component, allowing exceptions to propagate within the organization to a series of persons that may be involved in the identification and recovery actions. One human is always defined as being initially responsible for an exception, but can propagate the exception to other persons within the organization.

In the next section we identify and delimit the scope of our approach. Section 2 overviews related work. In section 3 we describe the concepts necessary to identify exceptions and define recovery actions. Section 4 begins with a brief introduction to the OS platform selected to implement the proposed solution and continues with a description of how the identification, situation awareness, propagation and recovery mechanisms are implemented in the platform. Finally, the last section presents the actual status of the project and indicates future work directions.

2. Scope and Limitations of Ad hoc Interventions

Our approach is based on two fundamental assumptions: 1) the ad hoc interventions are carried out on a problem solving basis through a coordinated effort of all persons in the organization that are able to contribute; 2) the set of interventions permitted to users should be, in one way, sufficiently complete to solve the highest number of

cases in the best possible way and, in the other way, sufficiently correct so that the process proceeds under engine control and without errors after the interventions.

Clearly, the first issue is a matter of Computer Supported Cooperative Work (CSCW) and will be critical to the implementation of our framework. Even though this matter is not the main objective of this paper, the work developed by [14] gives important guidelines on how to improve support to human interventions during exception recovery.

The second issue represents in some way a trade off: the higher the latitude of intervention, the higher the probability to have inconsistencies in the WF engine. Our approach was to study a large number of possible interventions and later verify their correctness. Before establishing the correctness criteria, we will discuss the various perspectives that should be taken into consideration when analyzing WfMS.

[27] identifies the following WF perspectives: 1) control flow; 2) resource or organization; 3) data or information; 4) task or function; and 5) operation or application. According to the author's arguments we will also abstract from resource, task and operation perspectives.

The data perspective will be discussed in more detail since it is a matter of some controversy. Our approach also abstracts from the control dimension. In fact, one of the primary objectives of WfMS was to remove control flow dependencies over data structures [17]. We advocate that any data inconsistencies should be identified and dealt within tasks. Moreover, the ad hoc interventions should not be constrained by data dependencies, as they can be dealt afterwards at the task level.

Our approach recognizes that a solid theoretical ground is needed to identify proper ad hoc interventions that keep the soundness (as defined in [27]) property of a WF. Therefore our focus is on the control perspective.

The concept of soundness assures that for any case the procedure terminates properly, i.e., termination is guaranteed, there are no dangling references, and deadlock and live lock are absent.

The adopted correctness criteria is slightly different from [23]:

> *The ad hoc interventions should not introduce any inconsistencies or errors in running instances (e.g., deadlocks or live locks). The process should be able to terminate without any other interventions under WfMS engine control after the interventions are carried out.*

Finally, exceptions that can be anticipated can also be handled with some degree of planning and therefore are not the main objective of this work.

3. Related Work

Exception handling has been mainly approached with a systemic perspective. The foremost solutions were based on variations of the transactional mechanism used in database management systems. [32] has a good survey on the different methods used by this approach. Some recent solutions [3; 7; 16; 18] deal mainly with anticipated events. These approaches are very useful to increase the flexibility of WfMS by increasing their ability to adapt to different circumstances. However, a framework to

support human involvement in solving exceptions has never been proposed in this context.

[14] presents an interaction framework for WF enactment. This framework is mostly important for unstructured processes and falls more on the CSCW area than on the systemic perspective. We believe that this framework is also important to guide human interventions during ad hoc operations.

[6] has one of the most complete studies of exceptions supporting human intervention. Although the cooperation of different users in solving exceptional situations is considered a critical issue, a conceptual framework to guide such approach is not proposed. We also do not see any evidence of the application of some correctness criteria.

In [26] a comprehensive model is proposed to deal with all possible types of exceptions but, again, a framework to involve the users is unavailable. The interventions dealing with unanticipated events are not presented as well.

4. Exception Handling

The exception handling process is divided in two phases: 1) exception identification; and 2) ad hoc interventions necessary to restore the system to a coherent state.

The next section describes the mechanisms necessary to identify the different classes of exceptions. The data structure that describes an exception is also detailed. The following section describes the exception recovery model and tools implemented to perform ad hoc interventions.

4.1. Exception Identification

There are several ways to identify exceptions in WF, according to different perspectives that one may apply to the problematic situation (the reader may find some orthogonal criteria for exceptions classification in the related literature [5; 8; 20; 24]). In particular, one may consider a system perspective and assume that an exception triggers an exceptional event in the system. On the other hand, some types of exceptions cannot be identified by the system and must be triggered by humans or external applications [5; 13]. The following classes of exceptional events are defined:

- Data events – Identified within the task that generates an error condition. Data events, even though identified within a particular instance, can affect a collection of instances (e.g., detection of the same trip being booked twice for the same client).
- Temporal events – Triggered on the occurrence of a given time stamp. Temporal events may be further classified into: timestamps, periodic and interval. Timestamps occur when a completion date associated with a task is not respected; periodic events occur on a determined periodical sequence (e.g., every morning at 9:00); and interval events are associated to time constraints between

two tasks, e.g., the maximum time allowed after task 1 finishes before task n starts.

- WF events – Triggered during task or process start/end operations. Examples: a deadlock situation or a loop being executed more than expected.
- External events – These events are triggered from external sources. Example: a user cancels a given order.
- Noncompliance events – Triggered whenever the system cannot handle the process due to differences between modeled tasks and reality.
- System/application events – Triggered when the system is not able to recover from lower level failures, such as database or network failures (lower level failures are propagated as semantic failures [9]).

The post-functions defined by the WfMC [33] are used to identify the presence of data events upon completion of a given task. On the presence of a data error, the WF engine automatically triggers the event and instantiates the exception recovery WF.

We will now describe separately the identification mechanism for the three different classes of temporal events.

The method used to identify the *interval* class, is depicted in Figure 1 using the Petri net notation [28]. Assume that the WF designer would like to define a time interval constraint between tasks 1 and n in Figure 1.a. Figure 1.b shows how the WF specification has been changed to incorporate that constraint. If $task_n$ is executed before t1 fires, the constraint was respected and no temporal event is triggered. However, if t1 fires before $task_n$, a token is placed on p2 and the system triggers an exceptional event. The transition t2 implements the same task as $task_n$ and is inserted in the specification to assure that the WF execution will not stop on $task_n$ if a temporal event is triggered.

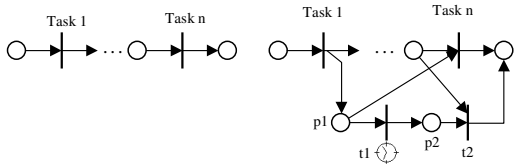

Fig. 1. Identification mechanism for the interval class

The firing of t1 will instantiate the exception recovery WF. The running workflow can be suspended or allowed to continue depending on the specific application.

For the *timestamp* class we use a similar scheme, where $task_1$ is the initial task and $task_n$ is the task identified in the timestamp. In this situation the timer is fired when the predefined date/time is reached. The exception recovery WF is instantiated as in the above example.

Figure 2 shows the implementation of the trigger mechanism for the *periodical* class. The original model is shown at the top where $task_1$ is the first task of the WF and $task_n$ is the last one. The place p1 and transition t1 where inserted to implement the *periodical* class. While the WF instance is running, the timer is also running. When the timeout is reached, one periodical event is triggered and the timer is

restarted. The timer stops with the firing of the last transition in the WF. Once again, the transition t1 instantiates the exception recovery WF.

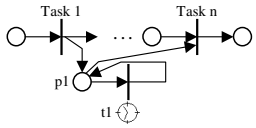

Fig. 2. Identification mechanism for the periodical class

To identify a WF event, a special condition must be inserted in the pre-functions or post-functions of every task that the WF modeler wants to monitor.

The external events are a particular category of events, because they cannot be detected by the system, as mentioned before. Thus, this type of event must be triggered by a human or external application.

The noncompliance events correspond to situations where the desired process either deviates from the model (by requiring some special treatment) or the model is not applicable to a particular context. In this type of situation the system requires some additional information regarding the model, i.e. additional tasks, tasks that should be modified or removed from the model, etc. Due to the intrinsic nature of these events, they are dependent of the specific context, which must be assessed by a human. Furthermore, these events may affect several tasks and processes.

Finally, the system/application events have characteristics similar to external events [19], although, in some circumstances, the exception may be automatically identified, e.g., the system is able to identify that a database server stopped without requiring human intervention.

Besides the trigger mechanism described above, we should now discuss the information that the system may associate to exceptional events. In this respect one should consider the following parameters:

1. Affected instance(s) –list of the affected WF instances.
2. Affected task(s) – identification of one or several tasks where the exception was identified. For instance, interval events and WF events are associated with one single task while data events may be associated with several tasks.
3. Data structures that characterize data events.
4. Expired timers, for temporal events in general.
5. Event categorization – classification of the event, as previously described.
6. A brief textual description of the event. This information applies to external events triggered by humans.
7. Model deviations – this applies to noncompliance events, and identifies a list of tasks that should be inserted, modified or removed.

We may also consider the following additional parameters:

8. Root cause – textual description, produced by a human, with the perceived root cause for an exception.
9. Person responsible – someone that may be responsible for the exception. This person may be selected by the system, from the list of persons associated to affected tasks, or selected by a human, as with the root cause mentioned above.

10. Impact – for every affected instance, the system may also provide information about deadlines and potential impact to the organization (based on metrics such as the diversity and number of affected tasks).

From the above list of items, the event categorization, affected tasks (at least one) and person responsible are mandatory.

It is now time to move forward from exception identification to recovery.

4.2. Exception Recovery

When an exceptional event is triggered, the system instantiates the WF recovery process modelled in Figure 3 and passes the several parameters identified in the previous section to the components described in this section.

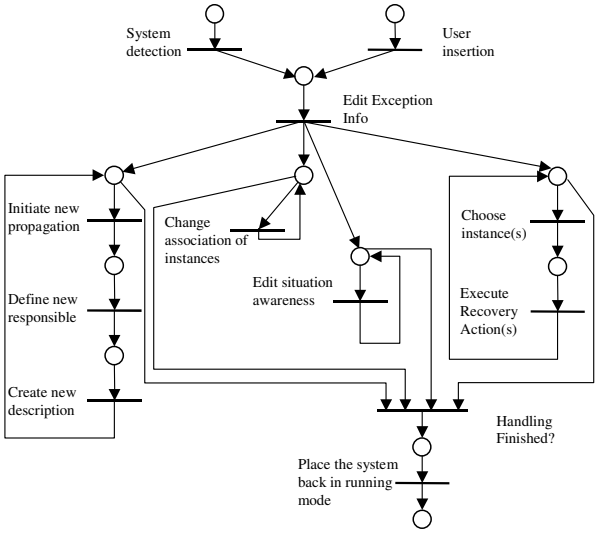

Fig. 3. Workflow Model for Exception Handling

There are two alternative ways to instantiate this process: either by *system detection* or by *user insertion*. They have been separated because these two tasks initialize the recovery process in different ways. The *system detection* task is used with the following event types: data, temporal, workflow, and system/application events. The *user insertion* task is used with external, noncompliance and system/application events (note that system/application events may be either identified by the system or by a human).

In both cases, the person responsible must have been identified, because that is the person who will be requested to execute the next action in the WF recovery process: *edit exception info*.

The purpose of this task is to specify some event parameters that the system was not able to specify, or should be redefined by a human (because human knows more

about the context). E.g., the root cause falls in the first case, while the list of affected instances and person responsible fall in the second case. This task is supported by the Exception Information Component, which has a User Interface (UI) and implements the several mechanisms necessary to identify exceptional events and interface with the WF engine (e.g., access timers and process variables, or obtain the list of affected instances).

After this task the system enters in four parallel threads:

- *Exception propagation;*
- *Affected instances;*
- *Situation awareness; and*
- *Apply recovery actions.*

The *exception propagation* task allows the person responsible to propagate the event to one or several persons. This task is supported by the Exception Propagation Component, which besides other functionality, implements the propagation mechanism by replicating exceptional events (i.e., no new events are generated, they are simply replicated) and stores the propagation history in a database.

During the *situation awareness* task, the person responsible can analyze previous propagations and all parameters associated to the exceptional event. Currently, the Situation Awareness Component implementing this functionality is basically a UI that transforms data provided by other components into a human readable format (although additional functionality has been conceived to relate this data with other information that may be available in organizational databases).

In the *affected instances* loop, the person responsible either selects WF instances one-by-one, if the scope is defined and limited, or selects the whole collection of WF instances from one process and associates to the exception. To select the affected instances one-by-one, the person responsible can navigate through the list of WF instances running in the WF engine. The user can also dissociate WF instance(s) previously affected to the exception. However, a WF instance cannot be dissociated if one of the recovery actions, apart from suspend/reinitialize, has been executed over it. The person responsible can also navigate through the list of processes running in the WF engine. The Situation Awareness Component also implements this functionality.

The *recovery actions* loop is where recovery actions may be executed on the selected WF instance(s). The person responsible first selects the WF instance(s) and then chooses one of the available actions. The Toolkit Component currently implements the following list of actions:

- *Suspend/reinitialize instance(s);*
- *Ad hoc refinement;*
- *Forward and backward jumps;*
- *Terminate instance(s);*
- *Move operation; and*
- *Ad hoc extension.*

Using the *suspend/reinitialize* action, the person responsible can suspend the execution of a specific instance(s). Later on, by issuing another action, the instance(s) can be set to the running state. During the suspended state no tasks can be initiated. However, the tasks that already have started are not aborted by the system. The per-

sons attached to those tasks are informed of the situation. These operations are not restricted since they do not affect the correctness criteria.

Using the *ad hoc refinement* action, the person responsible can perform a set of atomic activities from the list of standard WF activities, e.g., making a phone call, sending an email or writing a letter. The list of standard activities is currently small but expected to grow during system tests.

Still considering *ad hoc refinement*, another list is made available to the person responsible with all tasks defined in the affected process. The person responsible can then execute a task that was not yet executed, or repeat the execution of a task already executed. If a task is executed in advance and the user does not want to execute it twice, a marking mechanism is implemented that forces the task to be skipped when reached under model execution. The ad hoc refinement is not restricted. Based on [29], a parallel thread can be initiated, executing other tasks, while preserving the soundness of the final model. Furthermore, this is a valid transfer rule with no deadlocks and proper completion.

Backward jump skips to a previous step, while *forward jump* skips forward to another step in the WF instance.

As in [22], only *backward jumps* to actions in the history of the running instance will be allowed. However, since we advocate that the WF control evolution should be independent from WF data, we allow jumps to actions prior to loop iterations. We assume that tasks within and prior to the loop always assure the intended behavior of the loop control variables.

On the other hand, on the jump 1 in Figure 4, the system reaches a deadlock on S_5 because S_3 does not have any token. Only jump 2 is correct. To avoid this type of problem, the two rules of the following criteria must be satisfied: 1) the subnet starting at the destination place of the jump and finishing at the original place can be isolated (including every node in every branch leading from the start place to the end and every arc that finish or start on those nodes); 2) the isolated subnet is sound. The application of this rule follows from theorem 3, statement 3 in [27].

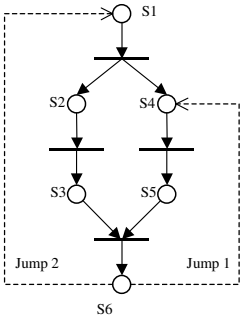

Fig. 4. Backward jump before AND-Splits

The two different ways to implement *forward jumps* are shown in figure 5. As in [22]: either the tasks in between are aborted (Figure 5.a) or executed in parallel with the tasks starting at the jump place (Figure 5.b).

If the tasks are aborted, the actual token is changed to the new place. A check must be done to assure that the system does not run into a deadlock or live lock situation.

Like in backward jumps, we restrict the jump to the condition that the subnet from the origin of the jump to the target can be delimited and are sound.

On the other hand, if the tasks are executed in parallel, an AND-Split is inserted on the transition before S_l (not show in Figure 5 for simplicity) and a task T_m (Figure 5.b) must be selected to synchronize the two parallel threads. The arc from T_n to S_n is removed and an AND-Join is inserted on task T_m with arcs from S_{m-1} and a newly created place S_k. This functionality requires modifying the model.

Figure 5 uses linear execution for simplicity. However, the operation will only be allowed if the subnets delimited from S_l to S_{n-1} and S_n to S_{m-1} (subnets as defined above) are sound. This statement can be proved from the properties of soundness.

To implement forward task execution (as described in [22]), the person responsible can use *ad hoc refinement* to execute the task and mark the tasks to be skipped (as mentioned before). This way, the task is executed during exception handling and skipped whenever reached during standard execution of the model.

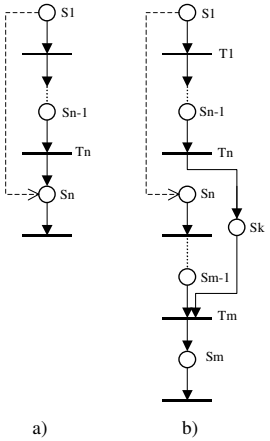

Fig. 5. Forward jump. a) abort tasks; b) parallel execution

To *terminate an instance* is to change a WF instance to the end state. No more actions will be executed on that instance.

The *move operation* moves a block in the process to a new location, keeping the remainder of the model unchanged. This change can only be executed if the final model is sound. Moreover, depending on the state of the instance, this operation can have different impact; hence, if there is more than 1 instance affected by this change, the dynamic change bug (as introduced in [11]) must be taken into consideration [29]. Our approach is to group instances according to their current state and apply different operations over each group.

Ad hoc extensions have a broader scope and a deeper impact on WF instance(s), since the person responsible can select an alternative path or choose a whole new model. On the alternative path scenario, we impose the restriction that only one thread is being executed on the instance. A check is made on the soundness property of the new path. If there is more than one instance affected, the change operation can be

applied only to those instances with tokens on the same places. Our approach is to group instances as in the previous situation.

For the new model situation, a correspondence must be established between every place where the current instance has a token and a place in the new model (called destination places from now on). To check consistency, a new place is inserted in the new model with an arc to an AND-Split. This new AND-Split will have arcs for every destination place. If this model is sound, the operation can be carried out. As in the previous situations, the dynamic change bug must be taken into consideration when several instances are affected. If the change cannot be performed to all instances, different change operations (for different target models) will be carried out. Some special care will have to be taken on backward jumps after this operation: no further backward jumps to destinations in the old sub-model should be allowed.

Once the recovery actions are executed and the system is back to a coherent state, the system executes the last transition, *place the system back in running mode*, and the exception handling is complete.

5. Implementation in the OpenSymphony Platform

The adopted OS is an open source platform that implements a WF engine, user and role validation, a timer component, persistence store of WF application data and Web interfaces. All components are developed in Java and run over a Servlet container. WF models are stored in XML files.

The next section introduces the platform and the two following sections describe the implementation of exception identification and recovery.

5.1. OpenSymphony Platform

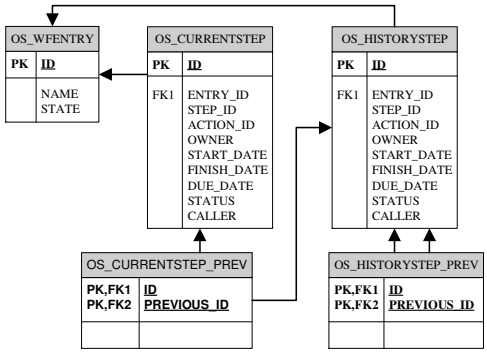

Fig. 6. OpenSymphony referential model

The "osworkflow" component of the OS platform implements the WF engine. This component stores the WF relevant data in a RDBMS. Figure 6 represents the complete set of tables and their relationships in the OS referential model.

The main table, OS_WFENTRY, after the workflow instance has been initialized, is shown in Figure 7. The ID field is the key for the WF instance, the NAME is the file with the model, and STATE indicates whether this instance is activated, suspended or completed.

ID	NAME	STATE
	...	
32	example.xml	Activated
	...	

Fig. 7. OS_WFENTRY table after the example initialization

When a new instance is created, the WF engine inserts a new row in the table with a generated ID field and the file name selected by the caller. After successfully execution of the initialization tasks, the field STATE is set to *activated.*

An example of the sequence of methods to create a workflow instance is:

```
Workflow wf = new BasicWorkflow(username);
long id = wf.initialize("example", initAction, mapIs)
```

The first method initializes the class and sets an internal variable with the username of the user logged in the system. The second method creates and initializes the new instance. The first parameter is the name of the XML file with the model, the `initAction` variable indicates the number of the action to be executed, and `mapIs` is a set of key to value pairs used by the action.

In the OS platform states are named steps. For every step there is a list with the actions available for execution. Figure 8 shows the typical hierarchical organization of a step with a listing of actions 1 to n. The elements of action 1 are also shown. The step in Figure 8 is the initial step for the WF and is named "initial actions". For the remaining steps in an OS model, the upper element has a NAME and ID that uniquely identify the step within the model (replacing the "initial actions" tag in the Figure 8).

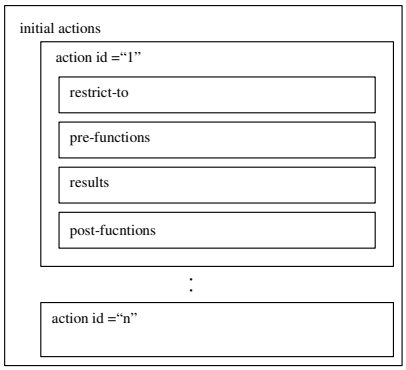

Fig. 8. Hierarchical organization of a step in the OS platform

To initialize the WF using action ID 1, the variable `initAction` must be equal to 1 in the `wf.initialize` method.

As represented in Figure 8, one action can contain four distinct elements: *restrict-to*, *pre-functions*, *results*, and *post-functions*.

The *restrict-to* element is composed by a series of conditions that must be evaluated to true to allow the execution of the action, e.g., only users that belong to a given role can execute that action.

After evaluating the conditions, the engine executes the *pre-functions*. They can implement tasks and set variables before the state transition takes place.

The next element is named *results* and is used to control the transition, i.e., the next step for the WF instance. Each element *results* can have 0 or more *conditional results* elements but must have at least one *unconditional result* element [21]. This structure can be compared to a "case" statement in a typical programming language where the element "case else" is mandatory. The first conditional element that is evaluated to true is executed. If none of the conditional elements is true the unconditional element is executed.

Let us assume that there are no conditional elements in action 1 and the unconditional element is:

```
<unconditional-result old-status="Finished"
        status="Run" step="1" owner="$(caller)"/>
```

The *unconditional result* indicates to the WF engine the number of the next step and a set of values to be stored in the database. This information is usually referred as WF relevant data [33] and will be described bellow.

After the transition takes place, the *post-functions* are executed (e.g., send an email to a user indicating that the action in the new step of the WF instance is ready to be executed).

To store the information about the current states of the various WF instances running on the system, the OS uses the table OS_CURRENTSTEP. Figure 9 lists the table field values after the successful initialization using action 1. The ID field is the key of this table that is automatically generated and ENTRY_ID is the foreign key to reference the WF entry table. The fields STEP_ID, OWNER, and STATUS reflect the attributes specified in the *unconditional result* element. The OWNER field is specified in the attribute owner and, assuming the username that triggered the initialization process was "Joao", is the value shown in Figure 9. The state assumed by the WF instance after the transition takes place is stored in the STEP_ID field and is defined in the attribute *step* (1 in the example). Finally, the attribute *status* specifies the field STATUS of the table and can assume any value.

ID	ENTRY _ID	STEP _ID	ACTIO N _ID	OWNER	START_ DATE	FINISH _DATE	DUE _DATE	STATUS	CALLER
. . . . .									
5	32	1		Joao	4/4/2004 11:50:33			Run	
. . . . .									

Fig. 9. OS_CURRENTSTEP table after the example initialization

The fields ACTION_ID, FINISH_DATE, and CALLER are set to null because they will be used when the next action, executed on step 1, is performed. The DUE_DATE field could have been used to set the desired due date for this task.

The *conditional and unconditional results* correspond to an OR-Split, i.e., various conditional results being tested and only one defining the next step means that the direction of the flow is chosen by the executed element. The AND-Split has a slightly more difficult definition that is out of the scope of this document. Nevertheless, if an AND-Split is executed, the table OS_CURRENTSTEP will have 2 entries for this instance.

After the transition takes place, i.e., the entries in the database are changed, the WF engine executes the *post-functions*. Then, the instance becomes idle until another action is performed over it. In the example, the WF is on step ID 1 waiting for any user-triggered action or any automatic action (the OS platform has a special type of actions, called automatic actions, which are automatically fired when the engine reaches the step where they are defined). The XML model files must have entries for every reachable step.

ID	ENTRY _ID	STEP _ID	ACTION _ID	OWNER	START_ DATE	FINISH _DATE	DUE _DATE	STATUS	CALLER
. . . . .									
5	32	1	3	Joao	4/4/2004 11:50:33	6/4/2004 15:30:45		Run	Joao
. . . . .									

Fig. 10. OS_HISTORYSTEP table after execution of action 2

Assume now, that action number 3 (defined in step 1 of example.xml) is later executed by username Joao on 6/4/2004 15:30:45. The row in Figure 9 is copied to the OS_HISTORYSTEP table and a new row is inserted in OS_CURRENTSTEP table reflecting the results of action 3. Figure 10 lists the table OS_HISTORYSTEP.

Figure 10 shows the fields ACTION_ID, FINISH_DATE, and CALLER with the values already settled and defined by the execution of action 3.

Figure 11 displays the state transition diagram for the engine and summarizes the above description.

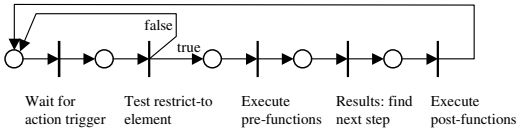

Fig. 11. State transition of the OS engine

To continue execution, the WF engine has methods to identify what are the available actions that a specific user can perform on a WF. These methods use the workflow ID to retrieve the actions defined in the model for the step.

Finally, the tables OS_CURRENTSTEP_PREV and OS_HISTORYSTEP_PREV identify the action executed before the current step and link the history of the tasks executed in the WF, respectively.

5.2. Exception Identification in the OS Platform

The process described in the previous section is used to create a new instance in the exception recovery WF, while the initialization of the exception related information is achieved through database utilities out of the scope of this paper. We will limit our description to the implementation of the exception triggering mechanisms.

The data events are implemented through the last pre-function element of the action. It identifies the presence of a data event and instantiates the exception recovery model. A post-function is also inserted to change the WF state to *suspended* if indicated in the data structure generated by the event. If the violated constraint is a generic rule, the other instance(s) that violates the constraint could also be identified and suspended if desired.

All temporal events are supported by the *Quartz* component provided by the OS platform, which implements a time triggering mechanism. Note also that a place in a Petri net is a step is OS and a transition is the set formed by the "pre-functions", "results" and "pos-functions. Their implementation in the OS platform is trivial, given the equivalence relation just mentioned, and therefore out of the scope of this paper.

In the case of an *end task*, a pre-function element is inserted in the action to identify the presence of an exceptional situation. For the *start task*, the functionality is implemented in a post-function. If the instance must be suspended, a post-function implements the functionality just after the transition takes place.

For the *start instance*, the test is carried out in the post-functions of the initial action and for the *end instance* a pre-function is inserted in the final action.

Finally, the system and application events are identified using the catch mechanism of programming languages. If during the execution of some code (condition, pre-functions, post-functions, or a user defined task) a non-caught exception construct from the program is raised, the code should instantiate the exception recovery WF. The decision whether the instance is suspended or not should depend on the code being executed, type of exception and application.

5.3. Exception Recovery in the OS Platform

To implement the exception recovery process, the model shown in Figure 3 was developed in an XML file, and the interfaces to propagate an exception, affect instances and edit descriptions were built using JSP to run over a Web environment.

As in the previous section, we will only describe the implementation of the various recovering tools used in the model.

The changes in the WF models of the OS platform are accomplished by editing the XML model file. A special method was developed to change the WF model used by a particular WF instance. This method will be used in various operations and changes the field NAME in the OS_WFENTRY table. A log entry is also generated for this operation. The description of the version system used in the new models is out of the scope of this paper.

In the tool *suspend/reinitialize instances(s)*, the field STATE of the WF OS_WFENTRY table is used. The *suspended* value in this field indicates that the WF instance cannot start any activity.

If a task started before the instance changes to the suspended state, a step transition can take place. The system should send messages to the person(s) executing manual tasks and to the supervisors of the automatic tasks.

In the *ad hoc refinement tool*, the list of standard actions is defined in a dedicated XML file. Some changes had to be made to the OS platform to support the execution of these actions within the scope of the instance. A WF model designer has to verify the modifications and inserts them in the model.

For the actions defined in the model of the running instance no special code was developed. These actions are listed to the user that can select the desired one.

To implement *forward and backward jumps* a new action is inserted in every step that uses the number of the destination step as a parameter.

To identify whether we are in the presence of a backward jump or a forward jump, the OS_HISTORYSTEP table is verified. If the destination step is in the table for this instance, we are in the presence of a backward jump, otherwise we must investigate the presence of a forward jump.

To allow a backward jump, the subnet, as defined in the exception recovery section, is identified. As this version of the system does not verify the soundness property of the models, we will restrict backward jumps to steps where the subnet only implements the sequence pattern, as defined in [30]. Later versions will implement the functionalities mentioned in the exception recovery section.

To investigate the presence of a forward jump, a simple algorithm is used to generate a tree of reachable steps from the current position. Once the destination step is found, we are in a presence of a forward jump. Any loop is iterated only once. A depth limit can be defined for complex models. If the step is reachable, the forward jump may be permitted. Again, as in backward jumps, only jumps in sequence patterns will be allowed.

If the user wants to implement a *forward jump* with parallel execution of the tasks between the actual step and the destination, the model must be changed. An AND-Split is inserted on the actual step and a task must be selected to synchronize the two parallel threads. An AND-Join is inserted on the task.

To *terminate an instance*, the field STATE of the OS_WFENTRY table is changed to *completed*.

The *move operation* requires model file modifications by a WF modeler to change the position of the task or block of tasks. Again, as the check for the soundness property is not implemented yet, this version will only allow changes of a block that only implements the sequence pattern and is moved in the limits of a branch that also implements the sequence pattern. If more than one instance with different step numbers are affected, the operation is divided into groups, as described in the exception recovery section. The change operation is implemented for every group. In some situations different models must be defined for different groups (when the instance state is between the previous position of the block and the new position), while in others only the matching between the original step number and the destination is different. E.g., assume a block was moved forward and there was no problem to execute the block twice. The instances that were executing a task in the middle of the block would become with the step number of the action that was positioned immediately after the block before the operation was carried out. This way

it is assured that these instances do not skip the tasks that are between the old and the new positions of the block. All other instances keep the step number.

In the alternative path of the *ad hoc extension*, the user chooses another WF model from a list with a predefined new trajectory for the remaining steps. As described in the exception recovery section, the instances are grouped according to their actual step. The new model must have one step number equal to the present step in the instances. The correctness of the new model is based on the same assumption of every model in the system: the WF modeler has enough knowledge to construct sound models.

In an *ad hoc extension* every step number of every active thread in the affected instance(s) must have been defined. Again, to overcome the dynamic change bug, different models can be generated according to the different combination of steps in the different instances.

In the two previous cases, if there are no available models the user contacts the WF modeler to develop a new one.

6. Actual status and future work

Two field tests and a simulation of the described system are in their initial phases. The field tests are being carried out in a Portuguese Port Authority and a design company, while the simulation tests are being carried out in a multinational automobile manufacturing company. The sizes of the two organizations involved in field tests are significantly different: the port authority has around 200 employees, while the design company has 10. By using different type of companies and different sizes we expect to understand how the system behaves in very different scenarios.

The completeness of the approach has to be validated, i.e., we have to test whether the implemented functionalities are complete enough to solve the exceptional situations that emerge in field tests. Some metrics used to evaluate the various ad hoc interventions will enable the selection of the most appropriate one for a given scenario in the future.

The impact on the number of models due to the number of instances affected by the exceptions will also be evaluated. The result of this evaluation will identify the need for the implementation of solutions for the dynamic change bug.

A test of the soundness property will increase the capability of the operations in various situations. This functionality will be developed in future versions.

On the other end, the growth of the standard list of actions used in ad hoc refinement will improve the system capability to deal with exceptions. The evolution of the list in the different scenarios will also be a matter of further research.

A log system that stores all the actions and propagations performed on every exception will be used to suggest strategies for similar situations. The mapping mechanism is a matter for later study.

We also expect to contribute to the development of the OS platform by increasing its flexibility to deal with exceptional situations.

References

1. Agostini, A., and De Michelis, G., 2000. Improving Flexibility of Workflow Management Systems. In: W.D. van der Aalst, J. Oberweis (Editor), BPM: Models, Techniques, and Empirical Studies. Springer-Verlag, pp. 218-234.
2. Agostini, A., and De Michelis, G., 2000. A light workflow management system using simple process models. Computer Supported Cooperative Work, 9(3): 335-363.
3. Casati, F., Ceri, S., Paraboschi, S., and Pozzi, G., 1999. Specification and Implementation of Exceptions in Workflow Management Systems. ACM Transactions on Database Systems, 24(3): 405-451. ACM Press.
4. Casati, F., Ceri, S., Pernici, B., and Pozzi, G., 1996. Workflow Evolution. Data and Knowledge Engineering, 24(3): 211-238.
5. Casati, F., and Pozzi, G., 1999. Modelling exceptional behaviors in commercial workflow management systems. Proc. IFCIS, CoopIS '99. UK, pp. 127-138.
6. Chiu, D.K., Li, Q., and Karlapalem, K., 2001. WEB Interface-Driven Cooperative Exception Handling in ADOME Workflow Management System. Information Systems, 26(2): 93-120. Elsevier Publishers.
7. Dayal, U., Hsu, M., and Ladin, R., 1991. A Transactional Model for Long-Running Activities. 17th VLDB'91. Barcelona, Spain.
8. Eder, J., and Liebhart, W., 1995. The Workflow Activity Model WAMO. Int. Conf. on Cooperative Information Systems. Vienna, Austria.
9. Eder, J., and Liebhart, W., 1996. Workflow Recovery. 1st CoopIS'96. IEEE International, Brussels, Belgium, pp. 124 - 134.
10. Ellis, C., and Keddara, K., 2000. A Workflow Change is a Workflow. In: W.D. van der Aalst, J. Oberweis (Editor), BPM: Models, Techniques, and Empirical Studies. Springer-Verlag, pp. 201-217.
11. Ellis, C., Keddara, K., and Rozenberg, G., 1995. Dynamic change within workflow systems. Proc. of Conf. on Organizational Computing Systems. ACM Press, Milpitas, CA, USA, pp. 10-21.
12. Ellis, C., and Nutt, G.J., 1993. Modeling and enactement of workflow systems. Application and Theory of Petri Nets. Springer-Verlag, USA, pp. 1-16.
13. Heinl, P., 1998. Exceptions During Workflow Execution. EDBT'98. Spain.
14. Jorgensen, H.D., 2001. Interaction as Framework for Flexible Workflow Modelling. Group '01. ACM Press, Boulder, Colorado, USA.
15. Kammer, P.J., Bolcer, G.A., Taylor, R.N., and Bergman, M., 2000. Techniques for Supporting Dynamic and Adaptive Workflow. Computer Supported Cooperative Work, 9(3): 269-292.
16. Klein, M., and Dellarocas, C., 2000. A Knowledge-Based Approach to Handling Exceptions in Workflow Systems. Computer Supported Cooperative Work, 9(3): 399-412. Kluwer Academic Publishers.
17. Leymann, F., 1997. Workflow-based applications. IBM Systems Journal, 36(1): 102-123.

18. Luo, Z., 2001. Knowledge sharing, Coordinated Exception Handling, and Intelligent Problem Solving for Cross-Organizational Business Processes. PhD Thesis, University of Georgia.
19. Mourão, H.R., and Antunes, P., 2003. Suporte à Intervenção de Operadores no Tratamento de Excepções em Fluxos de Trabalho. 4ª CAPSI. Porto, Portugal, pp. 29-42.
20. Mourão, H.R., and Antunes, P., 2003. Supporting Direct User Interventions in Exception Handling in Workflow Management Systems. 9th CRIWG 2003. Springer-Verlag, France, pp. 159-167.
21. The OpenSymphony project. Http://www.opensymphony.com2001, 20-04-2001.
22. Reichert, M., Dadam, P., and Bauer, T., 2003. Dealing with Forward and Backward Jumps in Workflow Management Systems. Software and Systems Modeling, 2(1): 37-58. Springer-Verlag.
23. Rinderle, S., Reichert, M., and Dadam, P., 2003. Evaluation of Correctness Criteria for Dynamic Workflow Changes. BPM '03. Springer-Verlag, Netherlands, pp. 41-57.
24. Saastamoinen, H., 1995. On the Handling of Exceptions in Information Systems. PhD Thesis, University of Jyväskylä.
25. Sadiq, S.W., 2000. Handling Dynamic Schema Change in Process Models. ADC 2000. 11th Australasian. IEEE International, pp. 120-126.
26. Sadiq, S.W., 2000. On Capturing Exceptions in Workflow Process Models. 4th International Conference on Business Information Systems. Poznan, Poland.
27. van der Aalst, W., 2000. Workflow Verification: Finding Control-Flow Errors using Petri-net-based Techniques. In: BPM: Models, Techniques, and Empirical Studies. Springer-Verlag, Berlin, pp. 161-183.
28. van der Aalst, W., 2002. Workflow Management. MIT Press, London, England.
29. van der Aalst, W., and Basten, T., 2002. Inheritance of workflows: an approach to tackling problems related to change. Theoretical Computer Science, 270(1): 125-203.
30. van der Aalst, W., Hofstede, A.H.T., Kiepuszewski, B., and Barros, A., 2002. Workflow Patterns, QUT Technical report, FIT-TR-2002-02.
31. Voorhoeve, M., and van der Aalst, W., 1997. Ad-hoc workflow: problems and solutions. Database and Expert Systems Applications, 1997, pp. 36-40.
32. Worah, D., and Sheth, A.P., 1997. Transactions in Transactional Workflows. In: S.K. Jajodia, Larry (Editor), Advanced Transaction Models and Architectures. Kluwer.
33. Workflow Management Coalition - Terminology & Glossary TC00-1011, 1999, Document Number WFMC-TC-1011, Issue 3.0. WFMC.