Handling Dynamic Changes in Decentralized Workflow Execution Environments*

Vijayalakshmi Atluri[1] and Soon Ae Chun[2]

[1] CIMIC and MS/IS Department, Rutgers University, USA
[2] Department of Computing and Decision Sciences, Seton Hall University, USA
{atluri,soon}@cimic.rutgers.edu

Abstract. Often, real world business processes are constantly changing and dynamic in nature. These runtime changes may stem from various requirements, such as changes to the goals of the business process, changes to the business rules of the organization, or exceptions arising during the workflow execution. Unfortunately, traditional workflow management systems do not provide sufficient flexibility to accommodate such *dynamic* and *adaptive* workflows that support run-time changes of in-progress workflow instances. Moreover, traditional workflow management is accomplished by a single centralized workflow management engine, which may not only be a performance bottleneck, but also unsuitable for the emerging internet-based commerce and service environments where workflows may span many organizations that are autonomous. In this paper, we propose a formal model for a *decentralized workflow change management* (DWFCM) that uses a rules topic ontology and a service ontology to support the needed run-time flexibilty. We present a system architecture and the workflow adaptation process that generates a new workflow that is *migration consistent* with the original workflow.

1 Introduction

With the internet-based commerce and service environment, business processes often cross organizational boundaries, as can be seen in *virtual enterprise* and *digital government* services. A workflow management system that supports these inter-organizational processes needs to address the following challenges: (1) The execution of these processes needs to be scalable and honor the autonomy of the participating organizations; (2) The definition of inter-organizational processes needs to be dynanmic and *ad hoc*, providing customization considering diverse user requirements and variations; (3) The workflow management should have sufficient flexibility to handle dynamic changes at run-time.

We have proposed a *decentralized workflow management model* (DWFMS) [1, 2], and a model for customized workflow generation (WFGM) [6] to address the first two challenges. In this paper, we present a *dynamic, decentralized workflow change management model* (DWFCM) based on ontologies. The ontologies

* This work is partially supported by the National Science Foundation under grant EIA-9983468 and by MERI (the Meadowlands Environmental Research Institute).

provide controlled vocabulary to specify different types of changes, and allow the system to automatically identify and incorporate necessary *migration rules*. The significant components of our DWFCM include: (1) *workflow context manager* (ConMan) that monitors changes in the user profiles, rules, and exceptions during task execution and (2) a *Self adaptor* that identifies the necessary migration rules, based on workflow composition rules, and performs the workflow migration such that changes are, as much as possible, local and minimal.

This paper is organized as follows. In section 2, we present different types and sources of dynamic changes. In section 3, we present a decentralized workflow execution model, followed by a workflow generation model in section 4. In section 5, our change management model is presented, followed by algorithms and methods to manage the dynamic changes in section 6[3].In section 7, our approach to automatic exception handling is presented. We discuss related work in section 8, followed by conclusions in section 9.

2 Dynamic Change Types

Dynamic changes to a workflow may occur due to (1) changes to the goals of the business process, that is, changes to the user profile, (2) unanticipated exceptions that arise during a task execution, and (3) changes to the business rules and regulations. Consider the following land development permit process:

An entrepreneur, Bill, wishes to build a car wash on a 5-acre vacant property that is 400 feet from the bank of the Hackensack River in Little Ferry, New Jersey. The zoning for this area is "neighborhood commercial," as designated by the New Jersey Meadowlands Commission (NJMC). In this example, the developer needs to perform tasks that are shown as solid lines in figure 1.

1. **Profile Change:** While a zoning certificate (task t_2 in figure 1) is under review, assume that Bill submits a request to add a storage mezzanine. This change in Bill's profile requires additional parking on the property, and assuming that the property does not have enough space for additional parking, variance-related tasks t_6 and t_7 need to be inserted, as shown in figure 1.
2. **Exceptions:** While reviewing the stream encroachment permit application (t_5), suppose that the engineers found some hazardous substances underground. According to the regulations, this would require a task t_9 (*notify to remove hazardous materials*) needs to be added before t_5 into the original workflow, as shown in figure 1.
3. **Rule Change:** Assume that while the development application review is under way, the waterfront regulation which requires a developer to obtain a Waterfront permit that applies to all the developments within 500 feet from the water has changed to apply to all the lots within 300 feet. As a result, the Waterfront permit task (t_8) needs to be deleted since this rule is no longer applicable.

[3] Due to the space limitation, we did not include the algorithms and proofs of theorems. See [5]

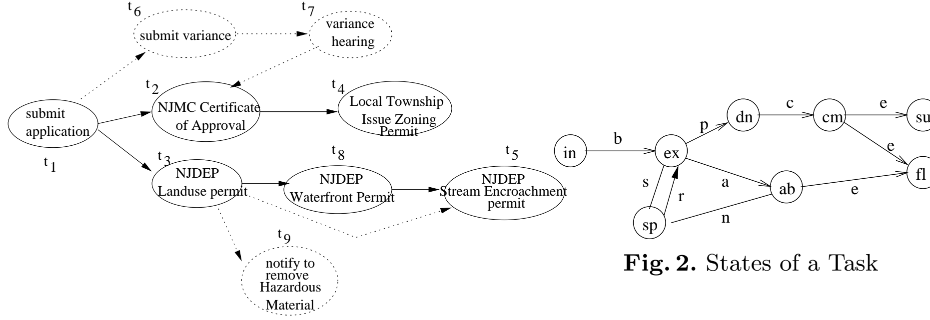

Fig. 2. States of a Task

Fig. 1. Examples of Workflow Changes

As can be seen from the above examples, the dynamic changes result in restructuring of the workflow, which can either be done as an insertion or a deletion of a task t_j. Such insertions can either be sequential or parallel to a task t_i that has not started its execution yet, or to a task t_k that has completed its execution. In some cases, the changes require to undo an already completed task and to redo it. If it is possible to undo or compensate for a completed task, then this change can be accommodated. On the other hand, change requests requiring to undo a completed task that does not have a compensating task will be rejected. Table 1 categorizes these various cases of structural workflow adaptation based on the type of operation.

The types of adaptation that can be accommodated are marked "yes," while the ones that cannot be accommodated are marked "no." The categories marked with "limited" indicate that the adaptation occurs if certain conditions are met, such as when redoing and compensating for a task is possible. The others, marked with "N/A," are cases that do not occur.

Restructuring of a completed part of the workflow does not occur in case of a rule change or of changes due to exceptions. This is, because changes in rules are always applicable from the time of change onwards. Exceptions are encountered only when a task is executing, so they do not affect the part of the workflow that has already been executed.

Table 1. Adaptation Types w.r.t uncompleted task t_i and completed task t_k

Operation	**Sequential**		**Parallel**
	before t_i	**after** t_i	
insert t_j	*yes*	*yes*	*yes*
delete t_j	*yes*	*yes*	*N/A*
	before t_k	**after** t_k	
insert t_j	*limited*	*limited*	*N/A*
delete t_j	*no*	*no*	*N/A*

3 Decentralized Workflow Model

A workflow and a task are formally defined as follows:

Definition 1. **[Workflow]** *A workflow W can be defined as a directed graph (T, D), where T, the set of nodes, denotes the tasks $t_1, t_2...t_n$ in W, and D, the set of edges, denotes the intertask dependencies $t_i \xrightarrow{x} t_j$, such that $t_i, t_j \in T$ and x is the type of the dependency.*

Definition 2. **[Task]** *Each task $t_i \in T$ is a 5-tuple $\langle$ A, Ω, Input, Output, EOutput $\rangle$, where A denotes the execution agent of t_i (denoted as $A(t_i)$), $\Omega = OP \cup PR$ is operations, Input the set of objects allowed as inputs to t_i, Output and EOutput are the set of objects expected as output from t_i and their expected values, respectively.*

Each task has internal structure represented as a state transition diagram (figure 2) with a set of states ST = {initial (in), executing (ex), done (dn), committed (cm), aborted (ab), succeeded (su), failed (fl), suspended (sp)}, and a set of primitive operations PR={b (begin), p (precommit), a (abort), c (commit), e (evaluate), s (suspend), r (resume), n (cancel)}.

The DWFMS enforces inter-task dependencies without a centralized WFMS. This model uses the notion of *self-describing workflows* (*Self*) and *WFMS stubs*. Self-describing workflows are partitions of a workflow that carry sufficient information so that they can be managed by a local task execution agent rather than the central WFMS. A WFMS stub is a light-weight component that can be attached to a task execution agent, which is responsible for receiving the self-describing workflow, modifying it and re-sending it to the next task execution agent.

When the *WFMS stub* at the first task agent receives a *Self*, it unpacks the information in *Self*, checks preconditions for its task and executes the task. Once the task agent finishes its task, it partitions the remaining workflow and generates *Self*s and forwards them to the subsequent task agents. This process continues until all tasks are executed and returned to the central WFMS stub.

4 Ontology-based Generation of Customized Workflows

Our earlier work [6] proposed a customized workflow generation model (WFGM). Workflow composition (generation) is achieved with the ontology of tasks, shown in figure 3, that captures the knowledge of available tasks and their relationships (definition 3), and the ontology of composition rules, shown in figure 5, that captures and organizes rules and regulations that prescribe how to compose tasks (definition 4).

Definition 3. **[Domain Tasks Ontology]** *A domain service ontology* SO *is defined as a set of services $\{s_1, s_2, ...\}$. Each $s_i \in$ SO is defined as a pair $\langle SA, SR \rangle$, where SA is a set of service attributes and SR= $\{rel_1, rel_2, ...\}$ a set of relationships. Each rel_i=$\{c_1, c_2, \ldots\}$ is a set of concepts that s_i bears the relationship to.*

Definition 4. **[Rules Topic Ontology]** *A rules topic ontology* RO *is defined as a set of rule topics* RO=$\{to_1, to_2, ...\}$. *Each $to_i \in$ RO is defined as a triple $\langle RA, Rel, R \rangle$, where RA denotes a set of attributes, Rel a set of relationships that to_i bears to other concepts, and $R = \{r_1, r_2, ...\}$ a set of composition rules associated with to_i.*

The relationships *has-subtopic*(to_i) define a set of subtopic concepts that are related to to_i, and *isa-subtopic*(to_i) defines a topic concept that is the super-topic of to_i.
Each composition rule is represented as a condition-action pair, which is defined below.

Definition 5. [Composition Rule] *Given a task* $t \in \mathrm{SO}$ *and a composition operation* $cop \in COP$=$\{$*insert,order,parallelize*$\}$, *a composition rule* r *is defined as a pair* $r = \langle c, a \rangle$ *where* c *is the condition and* $a = cop(\boldsymbol{t})$ *is the action.*

Example 1. r_1=$\langle$*distance(river)* $\leq$ *300 feet*, *insert(obtain Waterfront permit)*$\rangle$: r_1 states that if the distance from river is less than 300 feet, then insert task *obtain Waterfront permit* into the workflow and add *distance(river)* $\leq$ *300 feet* as the precondition for the task.

5 Decentralized Workflow Change Management Model

In this section, we present our Decentralized Workflow Change Management model (DWFCM) that allows a workflow executing under decentralized control to transparently adapt to run-time changes and exceptions so as to provide the needed flexibility. Our approach to managing dynamic changes includes (1) specify change requests, and exceptions, (2) identify tasks and operations for structural modification, and (3) apply migration of workflow partitions to new ones. Figure 4 shows the architecture of our dynamic workflow change management system.

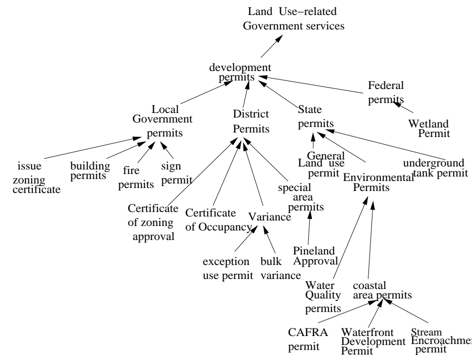

Fig. 3. Service Ontology

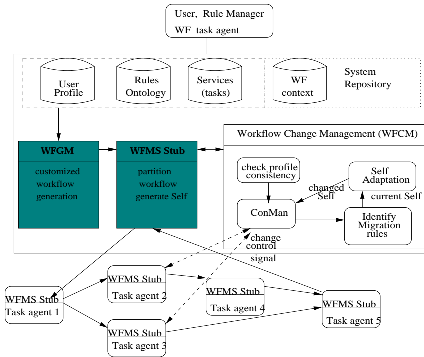

Fig. 4. The Dynamic Workflow Change Management System

5.1 Change Request

A change request can be made by a change agent, e.g. a user or a task agent or a rule administrator. A set of change request operators $ROP \subseteq \{$*insert, delete, change*$\}$ is provided for a change agent to place a request to insert, delete or change a profile, a rule or a workflow condition at run time. The following defines a change operand and a change request.

Definition 6. [Change Operand] *Given a Rules topic ontology* RO$=\{to_1, to_2, \ldots\}$*, a service ontology* SO$=\{s_1, s_2, \ldots\}$*, and a composition operator* $cop \in COP$*, a change operand co is defined as follows:*

- *If* $at{=}to_i \in$ RO *and* $v{=}to_j \in$ has-subtopic*(to_i), then a pair* $\langle at, v\rangle$ *is a change operand.*
- *If c is a condition and* $a{=}cop(\boldsymbol{t_i})$*, then* $\langle c, a\rangle$ *is a change operand.*

Definition 7. [Change Request] *Given a workflow* $W = (T, D)$*, a set of user preferences* $PRO = \{pro_1, pro_2, \ldots\}$*, a set of domain composition rules* $R{=}\{r_1, r_2, \ldots\}$*, and a set of change request operators* $ROP = \{$insert, delete, change$\}$*, a change request cr is defined as a tuple,* $\langle obj, \delta, co\rangle$ *where* $obj \in PRO \cup R \cup T$ *is an object to be changed,* $\delta \in ROP$ *is a change request operator and co is a change operand.*

Example 2.
- $\langle pro_{12}$, change, $\langle$ *business.type, restaurant*$\rangle\rangle$ where $pro_{12} = \langle$ *business.type, automobile service station*$\rangle\rangle$. This is a change request to change a business type from an automobile service station to a restaurant.
- $\langle r_{11}$, change, $\langle$*(coastal-development=yes)* $\wedge$ *(buffer-area* $\leq$*500ft), insert(Waterfront Development permit)*$\rangle\rangle$ where $r_{11} = \langle$*(coastal-development=yes)* $\wedge$ *buffer-area* $\leq$ *300ft, insert(Waterfront Development permit)*$\rangle$. This example illustrates that the current rule r_{11} is changed for a coastal area development to expand the Waterfront buffer area to 500 ft.

5.2 Workflow Context

Each workflow is associated with a sequence of contexts, which keeps track of the changes in workflow execution from the initial workflow up to the current time. Following formally defines a workflow context.

Definition 8. [Workflow Context] *A workflow W is associated with a sequence of workflow contexts, denoted as* $CN(W) = \{cn_0, cn_1, \ldots cn_n\}$*. Each context* cn_i *is defined as a tuple* $\langle ts, PRO, CR, WF\rangle$ *where ts is a timestamp for the context switch time from* cn_{i-1} *to* cn_i*, PRO is the user profile,* $CR \subset R$ *is the set of composition rules used in generating W, and WF is the set of self describing workflows* ($Self_c$) *active at time ts.*

We use $PRO(cn_i)$, $CR(cn_i)$ and $WF(cn_i)$ to denote the user profile, the set of rules applied in generating W, and the set of *self describing workflows* (Selfs) in currently executing task agents at context cn_i, respectively. Each *Self* in $WF(cn_i)$ contains the current task t_i, the precondition set (*PreSet*) for t_i, an *OutState* with the execution state and the output generated from the task t_j before t_i. We use $cn_j(W)$ to denote the jth context for workflow W.

A *context switch* from cn_i to cn_{i+1} occurs whenever a task agent A(t_i) finishes its task and forwards its execution environment and workflow partitions to the next task agents, $Self_{j_1}, \ldots, Self_{j_k}$, resulting in WF($cn_{i+1}$) = WF($cn_i$)$-\{$Self$_i\}$ $\cup$ $\{$Self$_{j_1}$, ... Self$_{j_k}\}$. Also, when one of the context components, $PRO(cn_i)$, $CR(cn_i)$ or $WF(cn_i)$ is updated, a context switch occurs.

The following defines change control signals that are used to communicate with the WFMS stubs at various task agents, to suspend, resume or cancel tasks.

Definition 9. [**Change Control Signal**] *Given a task change primitive* $cpr \in CPR = \{s, r, n\} \subset PR$*, a task* $t_i \in W$*, and its task agent* $A(t_i)$*, a change control signal signal is defined as a triple* $\langle cpr, t_i, A(t_i) \rangle$.
Depending on the type of change primitive *cpr*, a change control signal is referred to as *suspend-signal, resume-signal*, or *cancel-signal.*
Example 3. $\langle s, t_2, A(t_2) \rangle$ is an example of a suspend signal to notify the task agent $A(t_2)$ that it should suspend t_2.

5.3 Migration Rules

The adaptation process takes a change request and a set of workflow contexts, identifies a set of *migration rules* and applies them to transform the current workflow partitions. Our approach to identifying these migration rules is to use the regulatory ontology. The migration rules are extensions of compositional rules in a rules ontology that are used for generating a customized workflow. The migration rules are defined as follows:

Definition 10. [**Migration Rule**] *Given a set of composition rules* $R = \{r_1, r_2, \ldots\}$ *in the rules topic ontology* RO*, where each rule* $r_i = \langle c, a \rangle$ *is a condition action pair, and a set of migration operators* $MOP = COP \cup$ {*delete, redo, compensate*}*, where* COP = {*insert, order*}*, a migration rule mr is defined as follows: If* $r_i = \langle c, a \rangle$ *is a composition rule where* $a = cop(\boldsymbol{t})$ *with* $cop \in COP$ *and a task* $t \in T$*, then* $\langle c, a' \rangle$ *is a migration rule where* $a' = mop(\boldsymbol{t})$ *such that* $mop \in MOP$.

An example migration rule: mr_1 = ⟨ coastal water = tidally flowed waterway, delete(obtain Waterfront permit) ⟩ states that the task of obtaining a Waterfront permit needs to be deleted.

Definition 11. [**Workflow Migration Consistency**] *Let* $W = (T, D)$ *be the workflow before a change,* $W' = (T', D')$ *be the workflow after the change, and* R *be the set of composition rules in the rules topic ontology. We say that the changed workflow* W' *is* migration consistent *with* W*,*

1. *if* R *does not change, then* W' *satisfies* R *or*
2. *if* R *changes to* R'*, then* W' *satisfies* R'.

6 Handling Dynamic Changes

6.1 Ensuring Consistency in Profile Changes

The vocabulary used in the change operand specification may vary widely. In our approach, we avoid the individual variability and ambiguities in the operand specification by restricting the specification to only the vocabulary from the topic concept nodes of the composition rule topic ontology as shown in figure 5. For example, when a user wants to change his development location from "Neighborhood Commercial zone" to "Park and Recreations zone," the change operand ⟨ *zone, Park and Recreation* ⟩ can be formed using the concepts from the rules ontology, i.e. *zone* and *Park and Recreation.* The use of the rules topic ontology also ensures that change requests are consistent. For example, the change of *location* from *Neighborhood Commercial zone* to *Park and Recreation zone* also

imposes changes to all the subtopic nodes. The neighborhood commercial zone-related attributes such as water quality and flood zone become relevant. The following defines the profile change consistency constraint. We have developed an algorithm, a theorem and its proof to ensure profile change consistency [5].

Definition 12. **[Profile Change Consistency]** *Let* RO $=\{to_1, to_2, \ldots\}$ *be the rules topic ontology, PRO be the user profile, cr be the change request and to_i be the topic attribute in cr. We say that the new profile PRO' is* profile change consistent *with PRO if PRO' contains to_i and all subtopic descendants of to_i.*

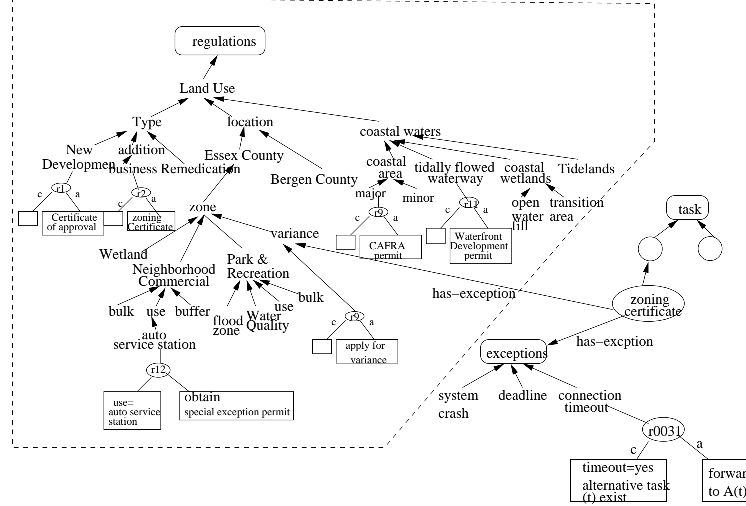

Fig. 5. A Rules Topic Ontology for land development

6.2 Ontology-based Identification of Migration Rules

In order to identify the necessary migration rules, Self Adaptor first examines the change request $cr = \langle obj, \delta, co \rangle$. For each type of change (user profile, rule, or workflow exception), a composition rule that satisfies the change operand co is selected from the rule ontology. If the attribute at in $co = \langle at, v \rangle$ matches a topic concept in the ontology, then the composition rules are evaluated against v. The composition rules that satisfy co are identified and converted into migration rules. We describe below how the migration rules are identified for each type of change request.

Case 1: Profile Change and Workflow Condition Change: If the change operation δ in $cr = \langle obj, \delta, co \rangle$ is 'insert', Self Adaptor first identifies a topic node in the rules topic ontology RO that matches the attribute at in the change operand co=$\langle at, v \rangle$. If there is any rule associated with the topic node, the rule is inserted into the set of insert rules. It does the same for all the subtopic nodes under this topic node to meet the profile change consistency constraint defined in definition 12.

When the change operation is 'delete,' Self Adaptor identifies the topic node, as in the case of insert, and applicable rules are added into a set of delete rules.

However, Self Adaptor checks if the task to be deleted has already finished its execution. If this is the case, it checks whether there is any compensating rule or task, and adds that to the insertion rules.

When the change operation is 'change,' Self Adaptor needs to identify an existing profile attribute value pair to be replaced. It then identifies insertion rules using the new profile values, as in the case of 'insert,' and it identifies deletion rules using old profile values as in the case of 'delete.' Then these sets of insertion and deletion rules are returned as a set of migration rules. For example, in figure 5, the user can request a change in his profile $\langle\ p_{12}$, delete, $\langle$ use, automobile service station $\rangle\rangle$ that was previously assumed. In this case, the system finds a node that matches the topic *use*, and the regulatory rule that is relevant to this topic ($r_{12} = \langle$ c, insert(t) $\rangle$ in the figure) is identified, where t is to obtain a special exception permit. Thus, the migration rule $\langle c,\ delete(t)\rangle$ is identified and inserted to be used to transform the current workflow partitions into new ones.

Similarly, the workflow condition change request is made by a task agent with cr=$\langle t_2$, *insert*, $\langle$*toxic waste site, yes*$\rangle\rangle$ to add a toxic waste site related task. In the same fashion as above, the composition rule related to the topic "toxic waste site" in the rules ontology is added as a migration rule.

Case 2: Rule Change: When a rule change request cr=$\langle obj, \delta, co\rangle$ is made, the change operand is in the form of a condition action pair, co=$\langle c, a\rangle$. In case of a rule to be inserted, Self Adaptor checks if the existing profile $PRO(ccn)$ satisfies the condition c in the changed rule. If the condition is satisfied and a is an insertion of a task t, then it checks if t is already in the current workflow. If such a task t exists, then Self Adaptor checks if it was already executed. If t was already executed, then no migration is necessary and the rule is not inserted into the set of migration rules. This avoids redoing the task t twice. Otherwise, it inserts the rule into the insertion migration operation. If there is no profile that satisfies this new rule, then there is no need to transform the workflow. The changed rule simply does not apply to the current workflow.

6.3 Self Migration

Once the migration rules are identified, Self Adaptor applies these rules to workflow partitions in $Self_c$. We call this migration process *Self adaptation* or *Self migration*. The Self migration process considers different types of migration operations, *compensate, redo, order, insert*, and *delete*. Every change is either adapted in the workflow generating the modified Self(s), denoted as $Self_{new}$, or a flag to reject the change is returned to ConMan. ConMan sends signals to the appropriate task agents to either resume or cancel $Self_c$ or restart execution with $Self_{new}$.

The following describes Self migration. Let the current workflow partitions be $\{P_1, P_2, \ldots\}$ and a set of migration rules be $MR = \{mr_1, mr_2, \ldots\}$, where each $mr_i = \langle c, a\rangle$ is a condition action rule with $a = mop(t_c)$. Self Adaptor goes through each migration rule in MR and applies migration operation mop to W in the applicable current partition P_i. If a *point-of-no-return* signal is returned during the migration rule identification process, the change is rejected. First,

Self Adaptor looks at all the *compensation* rules, and puts them into a partition in P_c. Then, it goes through the *redo* rules, and puts them into another partition P_r.
The other migration rules are subsequently incorporated into partitions P_i in the current $Self_c$. Each time when a migration rule applies to P_i, the partitions are updated to P^i_{new}. For an insertion rule (e.g. insert(t)), Self Adaptor checks if the task to be inserted has any ordering relationships to be met (e.g. order (t, t') or order(t', t)). If there is such a t', then t will be inserted into all the partitions with t'. For a deletion rule, Self Adaptor considers whether it is the first task, last task or a task between two tasks, and incorporates the rule in all the applicable partitions. Once the current partitions are updated with insertion and deletion migration rules, compensation and redo partitions are combined together, P_c followed by P_r. This combined partition is attached into one of the updated current partitions, P^i_{new}. The updated partitions are returned to ConMan for resuming the execution.

6.4 Workflow Context Management (ConMan)

When ConMan receives a change request for a workflow W, it sends suspend control signals to WFMS stubs that are currently executing W. Upon receiving a suspend control signal, t_i at $A(t_i)$ makes a transition from its current state to suspended (sp). When ConMan receives an acknowledgment from the currently active WFMS stubs, it invokes the migration process, *Self Adaptor*, to handle the change in the current self-describing workflows, $Self_c$ in $WF(ccn)$. The updated self-describing workflows $Self_{new}$ in WF_{new} are sent to appropriate task agents to resume the execution with updated self-describing workflows.

7 Automatic Exception Handling

Our approach to dynamic changes uses a change request initiated by a change agent, such as a task agent, the user or the rule administrator. This section provides an approach how the change request is automatically formulated by a WFMS stub, and submitted to ConMan for adaptation. In order to achieve the automatic identification of change needs and to formulate a change request automatically, a WFMS stub has to recognize whether the current workflow execution condition is considered an exception or not.

Recall from section 3 that in our workflow model, a task t is modeled as $\langle A, \Omega, Input, Output, EOutput \rangle$. When there are discrepancies between the actual output produced in *Output(t)* and the expected output *EOutput(t)*, the WFMS stub recognizes that there is an exception. In order for the WFMS stub to handle these exception, the WFMS stub suspends its execution, and notifies ConMan about the exceptions. ConMan sends suspend signals to active task agents, while the exception is being handled.

For exception handling, our approach uses the task ontology. Each task in the task ontology has a set of attributes and relationships. One relationship that a task may have is *has-exception*, which points to a topic node in the rules ontology. In other words, each task may have associations with specific exception-related

rules topics. Figure 5 shows that a task (*obtain zoning certificate*) has exception rules that are related to the topic *variances*. The task also has other generic exceptions, such as deadline or timeout, with a relationship link *has-exception* to a rule topic *exception*. These other exception rules, related to external exceptions, such as system connection failure within timeout period, are also considered.

The WFMS stub formulates a change request *cr* by comparing the *Output* and *EOutput*. If the output is different from the expected output, or an output parameter and its values are not expected, then a WFMS stub sets *cr* $=\langle t_i, insert, co\rangle$, where *co* is the pair $\langle$ output parameter, unexpected result $\rangle$. For example, the *toxic waste site* was not in the expected output, but occurred in the output. This is an exception that will be incorporated into the change request *cr*. Once the change request has been formulated, it is sent to ConMan and the exception handling process begins. ConMan first locates t_i in the service ontology and follows the link (relationship) *has-exception*(t_i) to locate exception-related rules. If *co* satisfies the condition of a rule r, then the action of r is added as a migration rule, and the Self migration begins.

8 Related Work

Several techniques for handling dynamic changes and exceptions in workflow management systems exist. Some approaches address correctness issues in conjunction with dynamic structural changes [7, 3, 10]. They are concerned with dynamic changes of the control flow and the handling of in-progress workflow instances when the schema is modified. Some *ad hoc* changes are handled by late modeling strategies, i.e. the workflow definition contains incomplete nodes that can be fully specified at run-time [4].

In HEMATOWork [9] the system tries to achieve automatic decisions on which instances need to be changed and what modification actions are needed using a rule knowledge base. It is different from our approach in that they do not utilize the hierarchical conceptual organization of rules. In [8], exceptions in collaborative processes between contractors and subcontractors in Electronic Commerce are handled with a generic process type hierarchy (called *process template*). Each template type is annotated with characteristic exception types. Every exception type has an associated knowledge base entry that provides information for what situations it is critical, and how it can be handled. This approach is close to our approach with respect to its unexpected exception handling. They do not address explicit change requests.

9 Conclusions and Future Research

We have provided an ontology-based framework for Decentralized Workflow Change Management (DWFCM), called "Self Migration," for automatic run-time adaptation of decentralized workflows (called Self), in response to changes in both the execution environment and user requirements as well as changes in rules and policies that govern the workflow. We also provide a mechanism for exception handling. Our change management framework provides a model to manage workflow contexts, called ConMan, that includes not only the workflow execution status and the partition locations, but also keeps track of a set of user

profiles and rules that are used in defining the customized workflow. Our model for Self adaptation allows 1) to express a change request, 2) to automatically identify, with the help of the rules ontology, migration rules necessary for accommodating changes, 3) to automatically apply these migration rules to generate a runtime customized workflow, and 4) resume and restart the execution.

Our approach is different from other studies on workflow adaptation, which assume the user's or a workflow designer's knowledge of the internal structure of the workflow or change knowledge. We intend to extend this work by pursuing the issues of workflow change authorization and accommodating simultaneous change requests from different sources.

References

1. Vijayalakshmi Atluri, Soon Ae Chun, and Pietro Mazzoleni. A Chinese Wall Security Model for Decentralized Workflow Systems. In *Proceedings of the 8th ACM Conference on Computer and Communications Security*, pages 48–57. ACM Press, 2001.
2. Vijayalakshmi Atluri, Soon Ae Chun, and Pietro Mazzoleni. Chinese Wall Security for Decentralized Workflow Management Systems. *Journal of Computer Security*, 2003. (in press).
3. F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Workflow Evolution. *Data & Knowledge Engineering*, 24:211–238, 1998.
4. Fabio Casati, Ski Ilnicki, Li jie Jin, Vasudev Krishnamoorthy, and Ming-Chien Shan. Adaptive and Dynamic Service Composition in eFlow. In *Proceedings of CAiSE 2000*, pages 13–31, 2000.
5. Soon Ae Chun. *Decentralized Management of Dynamic and Customized Workflow.* PhD thesis, Department of Mangement Science and Information Systems, Rutgers University, Newark, 2003.
6. Soon Ae Chun, Vijayalakshimi Atluri, and Nabil R. Adam. Domain Knowledge-based Automatic Workflow Generation. In *Proceedings of the 13th International Conference on Database and Expert Systems Applications (DEXA 2002)*, Aix-en-Province, France, September 2002.
7. Clarence Ellis, Karim Keddara, and Grzegorz Rozenberg. Dynamic change within workflow systems. In *Proceedings of Conference on Organizational Computing Systems*, 1995.
8. Mark Klein and Chrysanthos Dellarocas. A Knowledge-Based Approach to Handling Exceptions in Workflow Systems. *Journal of Computer-Supported Collaborative Work. Special Issue on Adaptive Workflow Systems*, Vol 9(No 3/4), August 2000.
9. R. Muller, B. Heller, M. Loffler, E. Rahm, and A. Winter. HematoWork: A Knowledge-based Workflow System for Distributed Cancer Therapy. In *Proc. GMDS98*, Bremen, September 1998.
10. Shazia Sadiq. On Capturing Exceptions in Workflow Process Models. In *Proceedings of the 4th International Conference on Business Information Systems*, Poznan, Poland, April 2000. Springer-Verlag.