Flexible Workflow Management in the OPENflow system

J. J. Halliday[1], S. K. Shrivastava[2] and S. M. Wheater[1,2]

[1]HP-Arjuna Labs., Newcastle-Upon-Tyne NE1 3DY, England

[2]Department of Computing Science, Newcastle University,
Newcastle upon Tyne, NE1 7RU, England

Abstract

Workflow management systems are required to provide flexible ways of managing workflows as the business processes they represent frequently require changes over time. Provision of flexibility features in workflow management systems is very much a research issue as workflow systems in use today have been found lacking in such features. Two approaches to achieving flexibility in workflows have been identified, namely flexibility by selection and flexibility by adaptation. Flexibility by selection is achieved by ensuring that there are a number of execution paths through the workflow process, such that key decision making points are well represented. This allows the appropriate path to be selected on a per-instance basis to take account of the prevailing circumstances. Flexibility by adaptation permits dynamic changes to workflows to include one or more new execution paths. This paper describes how flexibility is supported in the OPENflow distributed workflow system. In particular, it describes high level tool support for performing dynamic changes to a workflow. In OPENflow, dynamic reconfiguration mechanisms have been provided by making use of atomic transactions to add and remove one or more tasks and to allow the addition and removal of dependencies between tasks from a running workflow. Use of transactions ensures that changes are carried out atomically with respect to normal processing. An example application is described to illustrate flexible workflow management.

Key Words: workflow management systems, distributed systems, exception handling, flexible workflows, adaptive workflows, dynamic reconfiguration.

1. Introduction

Many organisations make use of workflow management systems for automating their business processes. Business processes frequently require changes over time. For example, a change in some business practice could require making changes not only to a business process definition (workflow schema), but also to instances of the corresponding schema currently in execution. It is therefore widely recognised that a workflow management system should provide flexible ways of managing workflows [1, 2, 3, 4]. Two approaches to achieving flexibility in workflows have been identified, namely flexibility by *selection* and flexibility by *adaptation* [4].

Flexibility by selection is achieved by ensuring that there are a number of execution paths through the workflow process, such that key decision making points are well represented. This allows the appropriate path to be selected on a per-instance basis to take account of the prevailing circumstances. Whereas flexibility by selection is typically achieved by design time modelling of the process, this may be supplemented by system support for late modelling. Late modelling involves leaving unclear or unknown portions of the workflow process schema undefined as design time. These 'black box' areas of the process are then defined at runtime, allowing for a greater degree of per-instance divergence.

Flexibility by selection is useful mainly where the requirements for flexibility in the workflow process can be identified in advance. This is insufficient for many systems, particularly those that implement a rapidly changing or long running business process. The flexibility by selection approach is also unsuitable for handling unforeseen exceptional circumstances that may arise at system or application level during the execution of a workflow process. Therefore, there is a need for *flexibility by adaptation*. Using this approach, the workflow process is altered to include one or more new execution paths. A workflow system requires additional functionality both in the execution engine and through user tools, to support flexibility by adaptation. Flexibility by adaptation may be further subdivided into *type adaptation* and *instance adaptation.* Type adaptation involves altering a workflow schema so that new workflow instances instantiated after the change is made, will use a new version of the process schema. This problem area is related to version control and is typically implemented by providing appropriate support in the schema repository part of the system. Instance adaptation concerns altering one or more running instances of a given schema. Such changes may involve only a specific instance (for example, to deal with exceptional circumstances arising in the business process), a given subset of the instances of the schema (for example, all the instances with a particular characteristic in their instance data), or globally to all running instances or the type (such as in the case of a change to the business process) It may also optionally involve updating the workflow schema in the repository so that the changes also affect future workflow instances.

Provision of flexibility features in workflow management systems is very much a research issue as workflow systems in use today have been found lacking such features. In addition, currently used systems tend to be monolithic in structure, making them unsuitable for deployment in distributed organisations. There is therefore much research activity on the construction of flexible, decentralised workflow architectures.

In this paper we describe how flexibility is supported in the OPENflow distributed workflow system. OPENflow has been designed and implemented as a set of CORBA services to run on top of a given ORB [5,6,7,8]. The system provides a distributed transactional execution environment to enable transactional and non-transactional tasks to be composed and executed as non-ACID 'extended transaction' workflows. OPENflow is unique in providing comprehensive support for flexibility in a distributed environment by selection as well as by adaptation. We highlight the system's main flexibility features here. In the former category, the system supports flexible task composition facilities that enable an application builder to incorporate alternative tasks, compensating tasks, replacement tasks etc., within an application to deal with a variety of exceptional situations. The system also supports "just in time" instantiation of parts of a workflow. This can be exploited to permit major changes within workflow applications to be performed efficiently. In the latter category, the system provides support for workflow adaptation at the instance level (also known as dynamic reconfiguration). In OPENflow, dynamic reconfiguration mechanisms have been provided by making use of atomic transactions to add and remove one or more tasks and to allow the addition and removal of dependencies between tasks from a running workflow. Use of transactions ensures that changes are carried out atomically with respect to normal processing.

This paper is structured as follows. In section two we summarise all of the flexibility features of OPENflow. Parts of the section two material has been derived from previously published papers on OPENflow and are included here to make the paper self contained. Sections three to five contain hitherto unpublished material and describe high level tool support for dynamic reconfiguration (section three), an example demonstrator application to illustrate flexibility features (section four) and related work (section five). OPENflow is a functioning research system; the features described here together with the example application of section four have all been implemented.

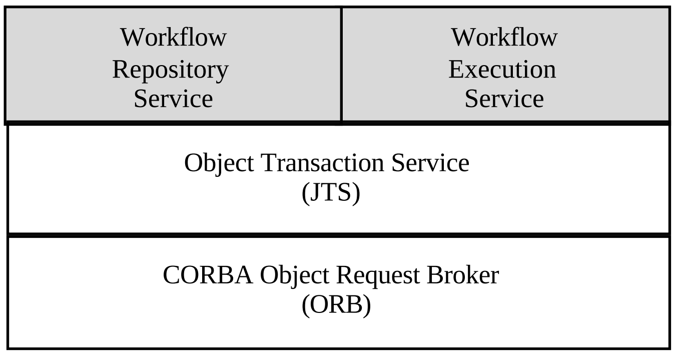
2. Support for Flexible Workflows

2.1. Overview of OPENflow

This section contains a brief description of those aspects of the workflow system that are necessary for understanding the flexibility features. The workflow management system has been constructed as two transactional services (written in Java), a workflow *repository* service and a workflow *execution* service (see fig. 1). These two facilities make use of the Java version of the CORBA Object Transaction Service (JTS); the implementation used is the JTS compliant version of the Arjuna distributed transaction system [9].

Workflow Repository Service: The repository service stores workflow schemas and provides operations for initialising, modifying and inspecting schemas. A schema is represented according to the task model described in the next sub-section in terms of tasks and dependencies.

Workflow Execution Service: The workflow execution service coordinates the execution of a workflow instance: it records inter-task dependencies of a schema in persistent atomic objects and uses atomic transactions for propagating coordination information to ensure that tasks are scheduled to run respecting their dependencies. The dependency information is maintained and managed by *task controllers*. Each task within a workflow application has a single dedicated task controller. The purpose of a task controller is to receive notifications of outputs of other task controllers and use this information to determine when its associated task can be started. The task controller is also responsible for propagating notifications of outputs of its task to other interested task controllers. Each task controller maintains a persistent, atomic object, *TaskControl*, that is used for recording task dependencies. A task controller is an active entity, a process, that contains an instance of a TaskControl object (however, to simplify subsequent descriptions, distinction between the two will often be blurred). In addition to TaskControl, the workflow execution service maintains two other key objects: instances of *Resource*, and instances of transactional objects *Task*. Objects whose references are to be passed between workflow tasks are derived from *Resource.* Task objects represent the workflow tasks which make up a workflow application (*Task*s are 'wrapper' objects to real application tasks). The most important operation contained within the Task interface is start, which takes as parameters: a reference to a TaskControl and a sequence of Resource references. The TaskControl reference is that of the controller of the task, and the sequence of Resource objects are the input parameters to the workflow task. TaskControl objects provide two operations (implemented as transactions): *request_notification* and *notification.*


Workflow Repository Service
Workflow Execution Service
Object Transaction Service (JTS)
CORBA Object Request Broker (ORB)
Figure 1: Workflow management facility structure.

The state transition diagram for a task controller is shown in fig. 2. The TaskControl object provides a *get_status* operation that returns its current state. During the initial setup phase, operations can be performed on the task controller to set inter-task dependency information. If task controller (tc_i) depends on output objects of some controller (tc_j) it must 'register' with tc_j by invoking *request_notification* operation of tc_j (a complementary, 'unregister' operation is available for deregistering). When the relevant objects of tc_j becomes available, tc_j invokes *notification* operation of tc_i to inform input availability.

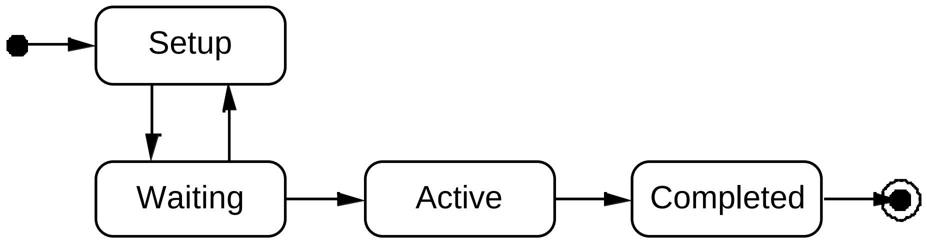


Figure 2: Task controller state diagram.

Once a task controller has been setup, it enters the waiting state. The waiting, active and complete states correspond respectively to the waiting, active and complete states of a task (see below). The task controller uses the *start* operation to start its task. Upon termination, a task invokes the notification operation of its controller to pass the results. Our system is able to support for dynamic modification of workflows because the TaskControl objects maintain the inter-task dependency information of an application and make this information available through transactional operations for performing changes [10].

2.2. Flexibility by selection

2.2.1. Flexible workflow composition

In order to explain the flexible composition features of OPENflow, we have to begin by describing briefly the task model. A task is an application specific unit of activity that requires specified input objects and produces specified output objects. An inter-task dependency may be a notification dependency indicating that a 'down stream' task can start only after an 'up stream' task has terminated (or started) or a data-flow dependency indicating that a 'down stream' task requires in addition to notification, input data from a 'up stream' task.

A task can be in one for three states: *wait*, *active* or *complete*; this is explained below. A task is modelled as having a set of *input sets* and a set of *output sets*. In fig. 3, task t_i is represented as having two input sets I_1 and I_2, and two output sets O_1 and O_2. A task instance begins its life in a *wait* state, awaiting the availability of one of its input sets. The execution of a task is triggered (the state changes to *active*) by the availability of an input set, only the first available input set will trigger the task, the subsequent availability of other input sets will not trigger the task (if multiple input sets became available simultaneously, then the input set with the highest priority is chosen for processing). For an input set to be available it must have received all of its constituent inputs (i.e., indicating that all data-flow and notification inputs have been satisfied). For example, in fig. 3, input set I_1 requires three inputs to be satisfied: two objects i_1 and i_2 must become available (data-flow dependencies) and one notification must be signalled (notifications are modelled as data-less input objects). A given input can be obtained from more than one source (e.g., three for i_3 in set I_2).

A task terminates (the state changes to *complete*) producing output objects belonging to exactly one of a set of output sets (O_1 or O_2 for task t_i). An output set consists of a (possibly empty) set of output objects (o_2 and o_3 for output set O_2).

Task instances manipulate references to input and output objects. A task is associated with one or more implementations (application code); at run time, a task instance is bound to a specific implementation.

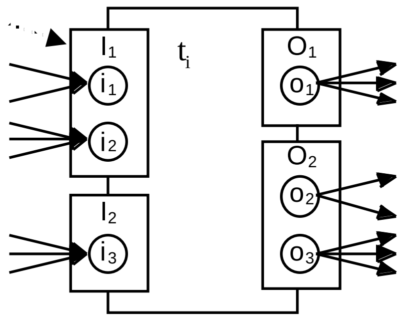


Figure 3: A task

The task model enables flexible ways of composing a workflow:

- *Alternative input sources*: A task can acquire a given input from more than one source. This is the principal way of introducing redundant data sources for a task and for a task to control input selection.
- *Alternative outputs*: A task can terminate in one of several output states, producing distinct outcomes. Assume that a task is an atomic transaction that transfers a sum of money from customer account *A* to customer account *B* by debiting *A* and crediting *B*. Then one outcome could be the result of the task committing and the other could be an indication that the task has aborted.
- *Compound tasks*: A task can be composed from other tasks. This is the principal way of composing a workflow out of other workflows. Individual tasks that make up an application can be *atomic* ('all or nothing' ACID transactions, possibly containing nested transactions within, with properties of: Atomicity, Consistency, Isolation and Durability) or *non-atomic*.
- *Genesis tasks*: A genesis task is a specialised form of compound task that acts as a place holder for a task structure, represented as a workflow schema. Its main purpose is to enable dynamic (on demand) instantiation of that schema. Structuring an application in terms of genesis tasks thus provides an efficient way of managing a very large workflow, as only those parts that are strictly needed are instantiated. Genesis tasks can also be utilised to specify workflow applications that contain recursive executions, that is a task structure whose execution will potentially cause the execution of its own structure as one of its sub-tasks.

The figure below shows a genesis task, t_3 which is contained within a compound task t_1. If task t_2 terminates producing the upper output set, this will cause task t_3 to be started; t_3 being a genesis task, this will cause the instantiation of the schema associated with t_3. The task structure associated with a genesis task can be determined at run time. This would allow, for example, task t_2 to change the task structure which is associated with task t_3, so causing major reconfiguration of the application's task structure. The use of genesis task in this way has the advantage that the configuring task just needs to name the schema associated with the genesis task in the workflow repository and there is no need to know the new structure or how to instantiate it.

Note that if the task structure associated with t_3 is the same task structure as the compound task t_1, this will produce a recursive structure resulting in the repeated execution of task t_2 (until task t_2 terminates producing the lower output set).

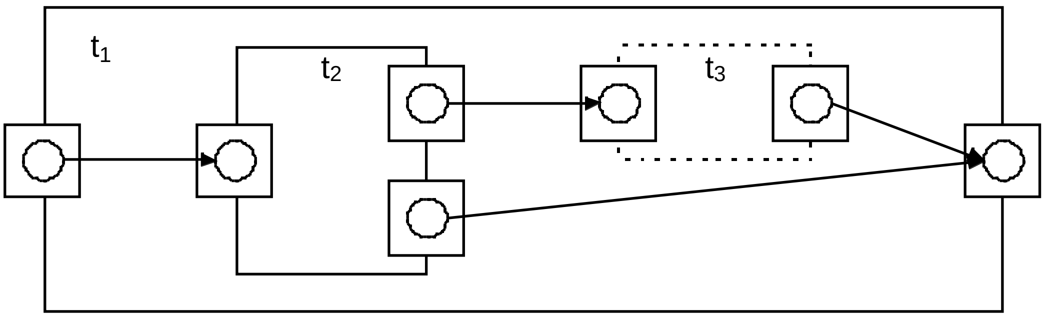


Figure 5: A genesis task.

2.2.2. Flexible deployment

A workflow is executed by deploying (instantiating) the corresponding workflow schema, which has the effect of creating sets of objects that control and represent tasks. A workflow schema (workflow definition) represents the structure of a business process in terms of constituent task definitions specifying details such as the task type, input data required for starting the task, sources of these inputs etc., as discussed earlier. In addition, it is also necessary to specify *organisation related information*, essentially stating *who* is responsible for carrying out that task (machines, workers) and *where* it is to be performed (e.g., the task 'answering customer queries' is to be carried out by a worker with the role 'help desk operator' located at the central headquarters). This organisation dependent information should be decoupled from the task definition, to ensure that changes to workflow definitions (respectively organisation structures) have limited effects, if any, on the organisation structures (respectively workflow definitions) [11].

Our system supports flexibility in deployment through a late binding mechanism implemented using *Task Factories* [12]. Tasks, the units of work from which a workflow process is composed, are created by invoking a method on a Task Factory object, passing it a set of *creation criteria* taken from the workflow schema. A workflow management system may contain a number of Task Factories distributed over the network according to the needs of the organisation. Each Task Factory may implement its own version of the task creation function, allowing for application or domain specific policies to be implemented easily. A basic Task Factory implementation requires only a single criterion, the Task Implementation class to instantiate. More advanced factories may allow the use of criteria that provide task allocation or placement policies. It is possible to chain Factories together. Under this arrangement, the Task Factory that initially receives the creation request examines the provided criterion and uses the information contained in them to decide on the Task Factory to which to delegate the creation request. For example, a creation criteria may be 'role=helpdeskstaff'. In this instance, the Task Factory may perform a LDAP query against an organisational directory server to resolve the role into an object reference for the Task Factory related to that role, to which the creation request would then be passed. This loose coupling of task definitions to resources of an organisation introduces a high degree of flexibility, allowing the environment in which the workflows execute to change without requiring alteration of the workflow schema. These aspects are discussed further in [12].

In addition to Task Factories, the system supports *Task Control Factories* for creation of TaskControl objects. An interesting feature of our system is that task controllers can be grouped in an arbitrary manner. For example, if dependability is crucial to the workflow application, then the task controllers can be placed on multiple machines so that the failure of a single machine will have a minimal effect on the progress of the workflow application. If the monitoring of the progress of the workflow application is more important than its dependability, then the task controllers can be grouped on the monitoring machine, thereby reducing communications overhead. In most cases the placement policy for the task controllers within the workflow application will be a compromise between these two extremes [7,8].

2.3. Flexibility by adaptation: Dynamic Reconfiguration

Our workflow system stores workflow schemas in a repository and it is possible to modify the structure of a stored schema; so type adaptation is supported. Here we focus on dynamic reconfiguration of a running workflow (instance adaptation). Below is the list of possible changes that can be performed on a workflow instance:

(a) The implementation bound to a simple task can be changed.
(b) Tasks can be added or removed from workflow instances.
(c) The constituent tasks of a compound task can be changed.
(d) Input alternatives can be added and removed from a task.
(e) The priority associated with input alternatives of a task can be changed.
(f) Output alternatives can be added and removed from a compound task.
(g) The priority associated with output alternatives of a compound task can be changed.
(h) The task structure associated with a genesis task can be changed.

These changes must be performed consistently, by which we mean respecting two conditions: (i) modifications to a workflow schema instance are carried out atomically (either all changes are performed or none) with respect to the normal processing activities; and (ii) the application is able to execute respecting these changes [8].

We use transactions to respect condition one. In addition, the following restrictions need to be observed to respect condition two.

R1: The implementation bound to a simple task can be changed, provided the task is in the wait state.
R2: The task structure bound to a genesis task can be changed, provided the task is in the wait state.
R3: Input alternatives cannot be added, removed or in anyway modified for tasks that are in state active or complete.
R4: Output alternatives cannot be added, removed or in anyway modified for a compound task that is in complete state.

To illustrate the use of this approach consider a simple application involving the processing of a customer's order, consisting of a compound task *processOrderApplication* which contains four constituent task instances: *paymentAuthorisation*, *checkStock*, *dispatch* and *paymentCapture* (see fig. 6). For the sake of simplicity input (output) sets are shown to contain at most one object each (an empty output set represents an aborted task termination). The *processOrderApplication* is started when an external input which is the customer's order is obtained. To process an order, *paymentAuthorisation* and *checkStock* tasks are executed concurrently. If both complete successfully then *dispatch* task is started and if that task is successful the *paymentCapture* task is started. If the *paymentAuthorisation*, *checkStock* or *dispatch* tasks fail, the *processOrderApplication* task will fail. This is represented, for all four tasks, by the production of the second output set, which contains no outputs.

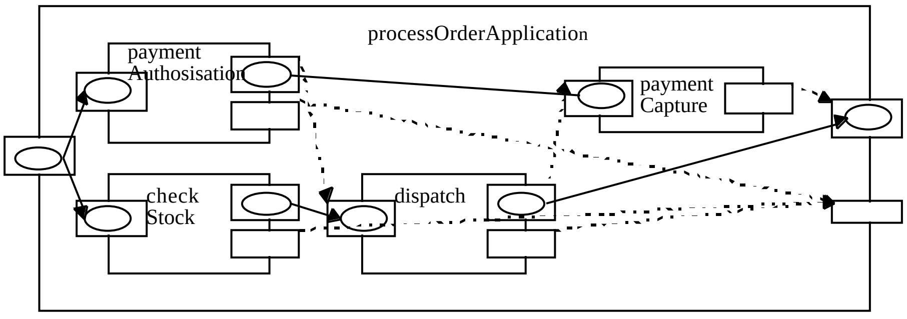


Figure 6: Customer order processing.

The internal structure of a compound task can be modified without affecting the tasks which supply it with inputs or use it for inputs. In this case it would be possible to change the payment and stock management policies, for example, causing payment capture even if the item is not presently in stock (a regrettable practice), or the addition of a task which could check the stock levels of the suppliers of the company, and arrange direct dispatch from them.

In fig. 7 the relationship between the task instances is shown that would result from the reconfiguration of the application to allow direct dispatch from a supplier. This reconfiguration involves the inclusion of an additional task, *directDispatch*, and additional dependencies between tasks, for example, the input of the order being needed by the *directDispatch* task. This reconfiguration can be performed dynamically, provided none of the restrictions *R1*, *R2*, *R3* and *R4* are violated. Here *R3* and *R4* are relevant; provided compound task *processOrderApplication* has terminated (*R4*), and *paymentCapture* has not started (*R3*) task *directDispatch* can be added.

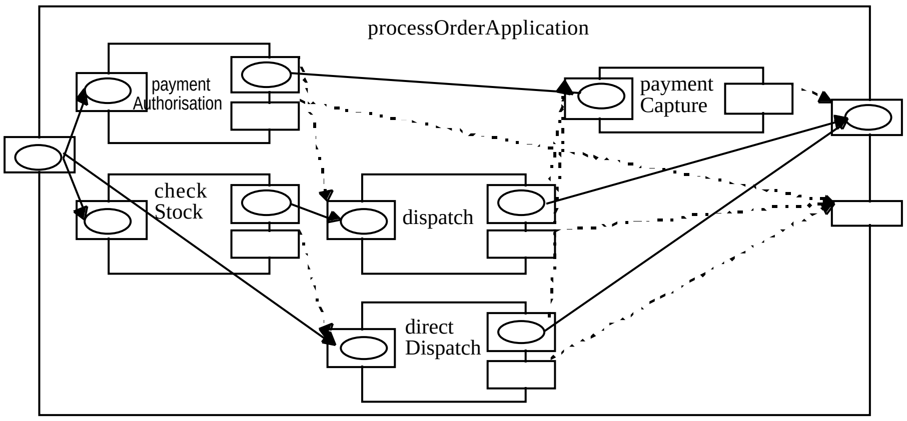


Figure 7: Reconfigured structure of process order workflow

3. Tool Support

We have developed a number of GUI based tools for workflow management. These tools have been developed as part of a service provisioning platform that uses OPENflow and a messaging middleware, and include tools for specification, deployment, monitoring and analysis [13]. The analysis tool uses model checking to detect liveness and safety properties of a workflow [14]. Here we concentrate on the GUI tool for dynamic reconfiguration and discuss a number of design issues.

Dynamic reconfiguration is made possible by allowing the Task Control objects that coordinate a given workflow process instance to be placed in a setup mode (fig. 2). Once in this state, dependencies between task controllers may be added, deleted or modified as desired. At the most basic level then, the requirements for a tool to support dynamic reconfiguration are the ability to identify and bind to the relevant TaskControl instances, to alter their state and to change the relationships between them. Consider the steps that would be required to reconfigure the *processOrderApplication* workflow application from the structure in fig. 6 to that of fig. 7:

1. The reconfiguration routine is passed an object reference to the task controller which is controlling the *processOrderApplication* compound task.
2. A transaction is begun, within which subsequent operation are performed (these operations check that restrictions *R1*, *R2*, *R3* and *R4* are being followed by checking the status of the appropriate task controllers, else the transaction is aborted).

3. An operation of the task controller of *processOrderApplication* is invoked to find the task controllers which contribute to the output of the task. This process is repeated until the entire structure of the *processOrderApplication* workflow application is found.
4. A task controller and task objects are created for *directDispatch*, the task being assigned to the task controller.
5. An operation on the task controller of *directDispatch* is invoked to specify that it requires an input from *processOrderApplication* and to specify that it requires a notification from *checkStock* (this will cause *request_notification* to be invoked on the task controllers of *processOrderApplication* and *checkStock*).
6. Similarly, an operation is invoked on the task controller of *processOrderApplication* to specify that it requires an output from *directDispatch* (this will cause *request_notification* to be invoked on the task controller of *directDispatch*).
7. Similarly, an operation is invoked on the task controller of *paymentCapture* to specify that it requires an input from *directDispatch* (this will cause *request_notification* to be invoked).
8. The transaction is ended, if the reconfiguration was successful the transaction will be committed, but if for any reason the reconfiguration was not completely successful (e.g., some *Ri* is violated) the transaction will be aborted. The effect of aborting the transaction is to recover/undo the effects of steps 3 to 7.

Clearly, high level tool support is necessary. A visual interface that allows for the graphical manipulation of workflow process instances provides a rich medium for the communication of dependencies and relationships between the constituent parts of a workflow process instance. Colour coding and text annotations of the glyphs used in the interface may also be used communicate additional information. Point and click operations may be employed, for example, to create or delete dependencies between task controllers.

As the core OPENflow system is written in Java, it was decided to use this language for our prototype dynamic reconfiguration tool. Java provides us with Swing, a standard cross platform library of components that form a GUI interface framework. In addition, the JavaBeans specification provides a standardised mechanism for writing interoperable components in Java. In order to implement a set of JavaBeans components that visually represent the state of a TaskControl we must address the problem of object distribution. In the Model-View-Controller (MVC) architecture commonly used in visual interface frameworks, the model object manages whatever data or values the component uses, such as the minimum, maximum, and current values used by a scroll bar. The view object manages the way the component is displayed. The control object determines what happens when a user interacts with the component, for example, what occurs when the user moves the scroll tab.

Although widely used in JavaBeans based applications, direct adoption of a MVC architecture is not feasible in our case as the task controller resides in another process space and thus cannot interface with the visual interface via bean events. The task controller is in any case not guaranteed to be written in Java, nor to support Java bean standards. To overcome this problem we create a local proxy object for the remote task controller. This object should not be confused with the automatically generated CORBA client stub for the TaskControl interface. Rather, it is a class that acts as an adapter layer between this stub and the world of JavaBeans. In addition to mapping function names to the beans standard, the adapter may be used to receive and generate bean events. Furthermore, we may utilise it to provide local caching of TaskControl state information. Such caching is necessary to produce an acceptably responsive GUI environment. If state information were not locally cached, screen redraw activity could potentially cause multiple CORBA calls across a network, resulting in very poor application performance.

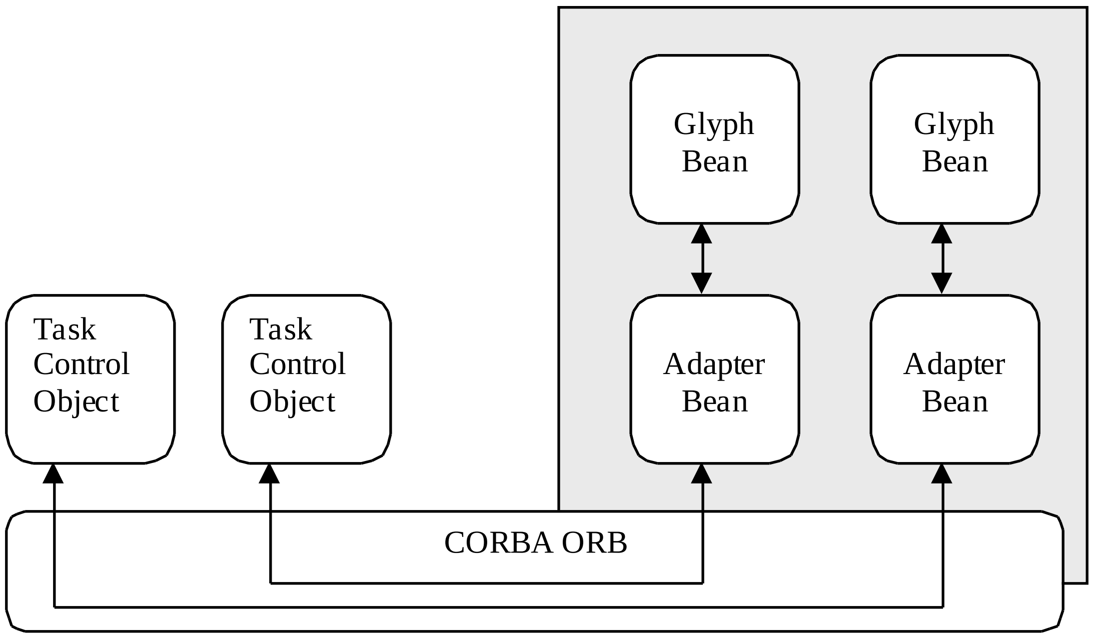


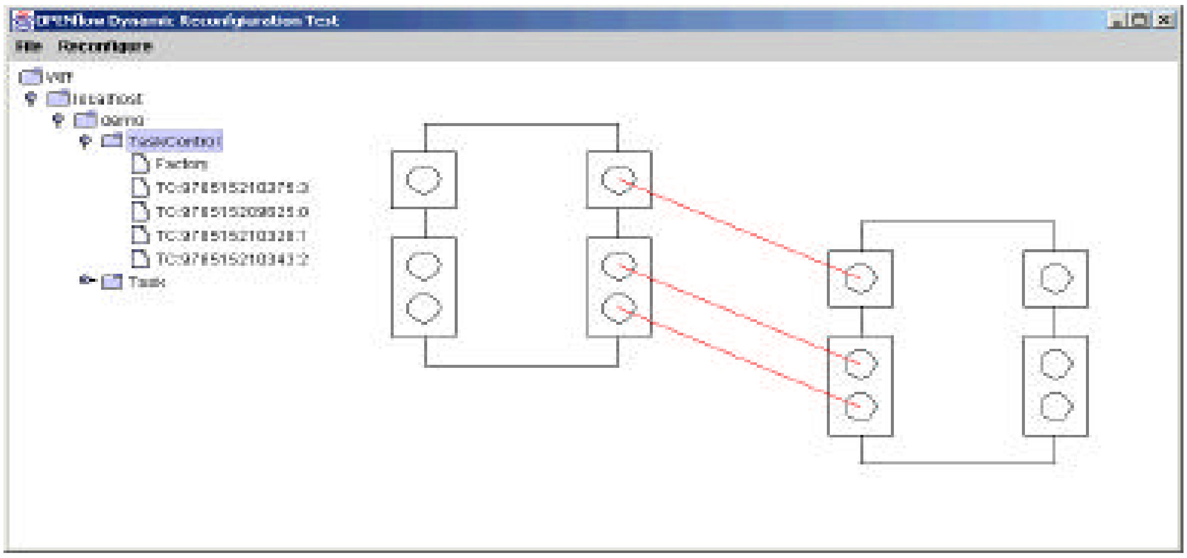
Figure 8: Object Architecture of the GUI-based Dynamic Reconfiguration Tool

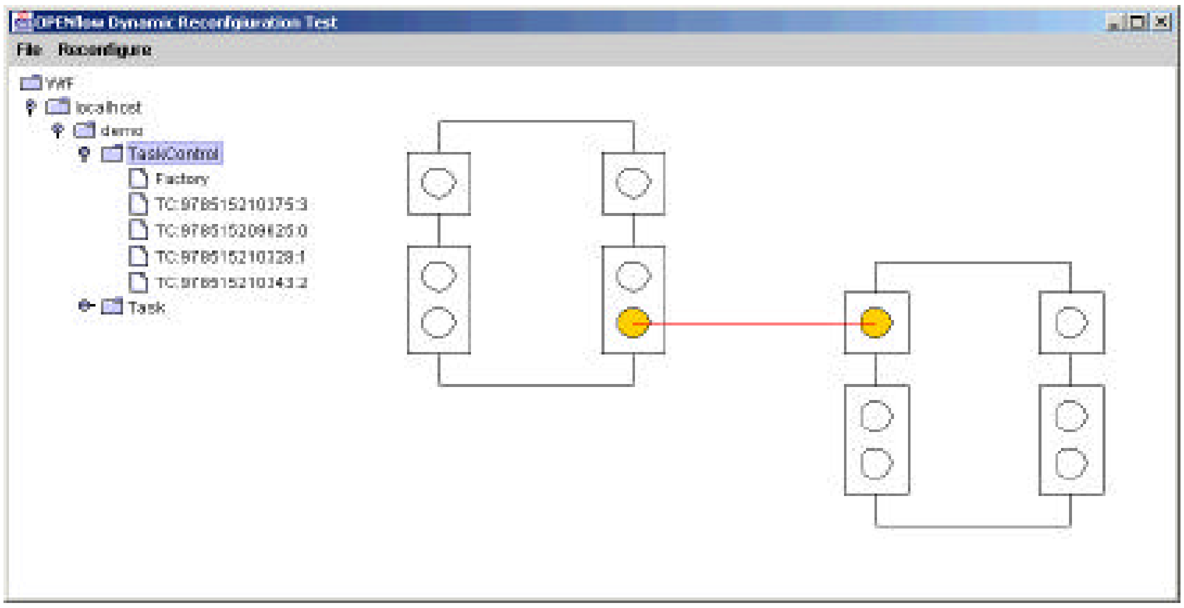
Whilst necessary, the introduction of locally cached state information for TaskControl objects gives rise to information consistency problems. Furthermore, it necessitates selection of write-through or write-back caching for state updates. Write-through caching would propagate all information updates to the remote TaskControl object synchronously. Write-back caching would allow these changes to be propagates lazily, perhaps on an user initiated or periodic basis.

The selection of a caching algorithm is a decision that is linked not only to performance issues, but also to transaction handling. As with all OPENflow components, the methods of the TaskControl interface must generally be accessed only within a transaction. The user of a dynamic reconfiguration tool may have their own requirements regarding the scope of these transactions. For example, they may be content to make each individual reconfiguration operation an isolated transaction. Alternatively, they may require support for manual transaction demarcation to allow a sequence of such operations to take place atomically. The situation is further complicated by a desire to minimise the disruptive effect of reconfiguration activities. Workflow process instances undergoing reconfiguration should be halted for no longer than is necessary. Likewise, other users should not be prevented from querying or updating the state of TaskControl objects just because they are being viewed in our reconfiguration tool. This problem is essentially one of optimistic versus pessimistic concurrency control.

If we allow reconfiguration operations to take place using locally cached state information and without requiring object locks to be held on the remote objects, initial performance will be excellent. However, the commit operation will trigger time consuming state synchronisation and may fail due to the inability to complete this procedure. If we obtain and hold object locks on any TaskControl object represented in the GUI throughout the lifetime of the user's session, we risk unnecessarily preventing other processes from utilising these objects for a prolonged period.

We have constructed a prototype tool for the dynamic reconfiguration of workflow process instances. This application is based on reusable JavaBean components. These include TaskControlGlyphBeans for the visual representation and manipulation of Task Control objects on the screen, and TaskControlProxyBeans for adapting between the CORBA and JavaBeans interfaces and for state caching. The tool integrates a CORBA name service browser component for the selection of TaskControl objects on which to operate. It allows new TaskControls to be added into a workflow processes and for the dependencies between constituent TaskControl objects to be added or deleted. Support for reconfiguring the number of input and output sets and objects in TaskControls, and supporting rebinding of Task implementations for TaskControls representing SimpleTasks is also available. The two screen shots shown here indicate how dependencies between two tasks can be changed.





4. Example

We have implemented a demonstrator application concerned with distributed firewall management that illustrates many of the features of the service provisioning platform mentioned at the start of section three. Here we will concentrate on workflow flexibility. Firewalls are security components interposed between the outside world and the internal networks of an organisation. All traffic passing through the firewall – web accesses, electronic mail, application transactions- is precisely identified, checked and allowed through or rejected depending the rules and regulations set down by the security policy. When large number of firewalls are involved (a number between 100 to 150 within an organisation is not uncommon), it becomes necessary to perform traffic analysis in a distributed manner, as each firewall generates hundreds of megabytes of data per day.

The demonstration simulates a virtual private network composed of 90 Intranets. The application is distributed on a linux cluster of 100 machines. The deployment of the entire firewall management configuration is performed under the control of a workflow. This is important, as deployment is a complicated process involving the deployment of hundreds to thousands of components over the network. Hence flexibility and dependability features of the workflow management system are extremely useful. Fig. 9 shows the compound task 'Deploy' that is in charge of launching in parallel the deployment of 90 firewall configurations (internal details of Deploy are not important here). This task takes as input the list of the configurations to deploy and gives as outputs the list of the deployed configuration instance handles, including the handles of configurations for which the deployment did not succeed. As a consequence, there are three possible outputs for such a task. The first one is triggered if all the configurations have successfully been deployed (*complete deployment*). The second one is triggered if some have failed (*partial deployment*). The last one is triggered if none succeeded (*no deployment*).

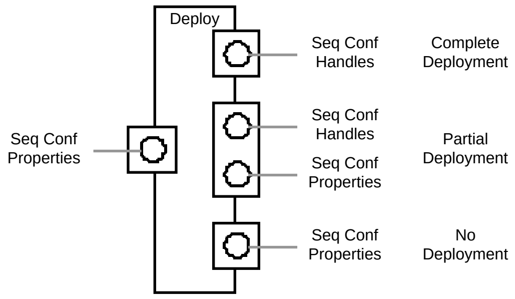


Figure 9: Workflow deployment task

One very simple deployment strategy is just to use the above task as a workflow; but this workflow does not handle failures in deployment. An improved workflow illustrating flexibility by selection is shown in fig. 10. A way to handle failures is to retry the deployment of the configurations that failed (perhaps the failures were due to temporary outages of network and or machines). To do so, a *Join* task has been added in order to merge partial results; the output sets of this workflow have the same meaning as that shown in fig. 9. It is still possible that the deployment of some configurations does not succeed. We assume however that this is a rare situation that requires manual intervention. Suppose that the system administrator determines that new resources have become available that could enable successful deployment of remaining configurations by executing a specific deploy task; the administrator does this by changing the workflow instance as depicted in fig. 11 (gray tasks).

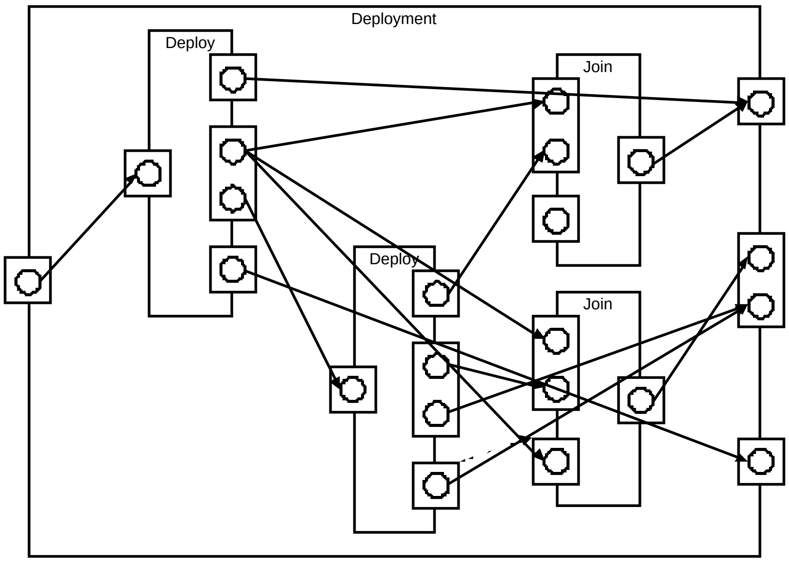


Fig. 10: Flexible deployment workflow

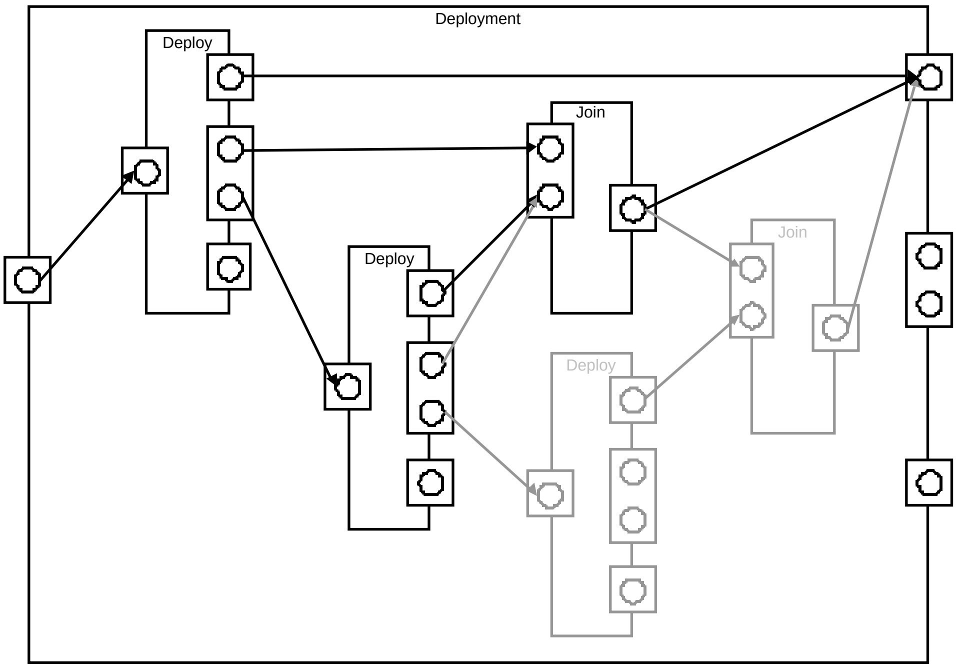


Figure 11: Performing changes to the deployment workflow

5. Related work

As stated in the introduction, a number of researchers have discussed the need for flexibility in workflow management; we have used the framework described in [4] for discussing the flexibility features of OPENflow. Casati et al. [2] describe a number of abstract operations that could be used for making changes to a workflow instance. These roughly correspond to the low level operations provided by TaskControl objects and are used by our GUI tool for performing change. Given that we use transactions for inter-task coordination, it is but natural to go a step beyond and use them for dynamic workflow modification. We are not aware of any other working workflow system that supports this form of functionality in a distributed environment. A number of distributed workflow systems offer support for exception handling (e.g. [15]). A few distributed workflow systems have appeared recently with facilities for dynamic change.

RainMan/RainMaker system has *sources* that generate service requests and *performers* that manage the servicing of the requests [16]. The PerformerAgent interface in RainMaker is implemented by both automated and human workflow participants and provides functions to create, run, suspend, resume and abort workflow tasks. At runtime, the same tool is responsible for interpreting the workflow schema and acting as a request source and coordinator of the process instance. It sends tasks to the appropriate PerformerAgents and interprets the next step in the workflow process based on the results it receives. This step-wise, interpretation based model allows for workflow processes to be reconfigured as they are running. However, unlike our system, no transactional support is available to enable changes to be carried out consistently.

WASA2 system [17] also offers support for the dynamic modification of workflow processes. Each workflow process instance is represented by a CORBA object. Complex workflow may be hierarchically composed of other workflows, leading to object trees. Each process instance may be reconfigured by freezing its execution, altering its internal state to represent the desired process definition, and then allowing its execution to resume. Where the workflow is composite, only the sub-workflow directly affected by the modifications need be halted. However, whilst the persistent state of the task objects allows crash recovery to be implemented, notifications between tasks remain non transactional. This gives rise to a number of problems, including issues such as the necessity of queuing messages for temporarily unavailable (crashed or frozen) workflow objects. Furthermore, reconfiguration support in WASA2 does not go much beyond the theoretical model summarised here and is the subject of ongoing research.

The METUflow2 system uses a workflow model designed specifically for dynamic adaptation to changes in its environment [18]. Its capabilities are focused mainly on instance level modifications. To facilitate these, each process instance is implemented as a distributed object that contains all control flow information and history. In order to modify a running workflow process instance, the user downloads a reconfiguration applet from the applet web server. The applet then uses the domain manager server to obtain a reference to the relevant workflow process object. Once bound to the instance, the user may make modifications using a graphical interface or using FLOW-ML, a purpose written language for expressing workflow modifications. Changes may optionally be propagated back to the workflow definitions library and to other running instances of the same type. In certain circumstances, the use of compensating tasks allows for running instances that have passed a critical point to be wound back in order to accommodate process modifications. However, whilst the individual tasks of a workflow process may be defined as transaction, there is no indication that the workflow infrastructure itself makes use of transactions for fault tolerance. This means that workflow processes may be left in an inconsistent state by client failure during reconfiguration activities.

The PROSYT system [19] takes a different approach to supporting adaptation in workflow processes. Rather than requiring the workflow process definition to be changed, the runtime engine permits the execution of individual instances to deviate from the workflow schema. A constraint language is employed to specify the nature and extent of the permitted deviations and the actions to take when they occur. The workflow engine tracks the deviations and offers support for reconciling the actual workflow state with the desired i.e. specified state. This may be accomplished either by altering the process specification such that the actual state is considered valid, or taking steps to alter the current state into one that is considered valid in the existing model. The author argues that approaches that require making changes to a workflow instance are complex to use and too elaborate when perhaps ‘minor deviations’ are required. Whilst our approach does require users to perform dynamic reconfiguration actions, we aim to hide much of this complexity through easy to use GUIs for many reconfiguration operations.

Acknowledgements

Work on OPENflow was supported initially by Nortel Networks and later on by ESPRIT LTR Project C3DS (project no. 24962) and UK Engineering and Physical Sciences Research Council. Luc Bellissard and D. Féliot played key roles in the development of the firewall demonstrator application; Mark Little was responsible for making the JTS compliant version of the Arjuna transaction system.

References

[1] C. Ellis, K. Keddara and G. Rozenberg, “Dynamic change within workflow systems”, ACM Conf. on Organizational Computing Systems (COOS 95), August 1995.

[2] F. Casati, S. Ceri, B. Pernici and G. Pozzi, “Workflow evolution”, Proc. of 15th Intl. Conf. on Conceptual Modelling ’96, Germany, October 1996, pp. 438-455.

[3] Y. Han, A. Seth and C. Bussler, “A taxonomy of adaptive workflow management”, ACM CSCW 98 workshop proceedings, ‘Towards Adaptive Workflow Systems”, Seattle, 1998.

[4] P. Heinl, S. Horn, S. Jablonski, J. Neeb, K. Stein, and M. Teschke, “A Comprehensive Approach to Flexibility in Workflow Management Systems”, Proc. Joint Intl. Conf. on Work Activity Coordination and Collaboration, WACC’99, San Francisco, Feb. 1999, ACM Software Eng. Notes, March 1999.

[5] F. Ranno, S. K Shrivastava and S. M Wheater, "A system for specifying and coordinating the execution of reliable distributed applications", DAIS'97, Distributed Applications and Interoperable Systems, eds: H. Konig, K. G. Geihs and T. Preuss, Chapman and Hall, ISBN 0 412 82340 3, 1997, pp. 281-294.

[6] F. Ranno, S K Shrivastava and S M Wheater, "A Language for Specifying the Composition of Reliable Distributed Applications", 18th IEEE Intl. Conf. on Distributed Computing Systems, ICDCS’98, Amsterdam, May 1998, pp. 534-543.

[7] S. M Wheater, S. K Shrivastava and F. Ranno "A CORBA Compliant Transactional Workflow System for Internet Applications ", Proc. Of IFIP Intl. Conference on Distributed Systems Platforms and Open Distributed Processing, Middleware 98, (N. Davies, K. Raymond, J. Seitz, eds.), Springer-Verlag, London, 1998, ISBN 1-85233-088-0, pp. 3-18.

[8] S. M. Wheater, S. K. Shrivastava and F. Ranno, “OPENflow: A CORBA based Transactional Workflow System”, Chapt. 15, Advances in Distributed Systems (S. Krakowiak, S. K. Shrivastava, Editors), LNCS No. 1752.

[9] M.C. Little and S. K. Shrivastava, “Java Transactions for the Internet”, 4th USENIX Conference on Object-Oriented Technologies and Systems, COOTS, Santa Fe, April 1998, pp. 89-100.

[10] S. K. Shrivastava and S M Wheater, “Architectural Support for Dynamic Reconfiguration of distributed workflow Applications”, IEE Proceedings – Software, Vol. 145, No. 5, October 1998, pp. 155-162.

[11] H. Schuster, J. Neeb and R. Schamburger, “A configuration management approach to large workflow management systems”, Proc. Joint Intl. Conf. on Work Activity Coordination and Collaboration, WACC’99, San Francisco, Feb. 1999, ACM Software Eng. Notes, March 1999, pp. 177-186.

[12] J. J. Halliday, S. K. Shrivastava, S. M. Wheater, “Implementing Support for Work Activity Coordination within a Distributed Workflow System”, Proc. of 3rd IEEE/OMG International Enterprise Distributed Object Computing Conference (EDOC’99), September 1999, University of Mannheim, Germany, pp. 116-123.

[13] S. K. Shrivastava, L. Bellissard, D. Féliot, M. Herrmann, N. dePalma, S.M. Wheater, “A Workflow and Agent based Platform for Service Provisioning”, Proc. of 4th IEEE/OMG International Enterprise Distributed Object Computing Conference (EDOC 2000), September 2000, Makuhari, Japan, pp. 38-47.

[14] C. Karamanolis, D. Giannakopoulou, J. Magee and S. M. Wheater, “Model Checking of Workflow Schemas”, Proc. of 4th IEEE/OMG International Enterprise Distributed Object Computing Conference (EDOC 2000), September 2000, Makuhari, Japan.

[15] G. Alonso and C. Hagen, “Flexible exception handling in the Opera process support system”, 18th IEEE Intl. Conf. on Distributed Computing Systems, ICDCS’98, Amsterdam, May 1998.

[16] S. Paul, E. Park and J. Chaar, "RainMan: a Workflow System for the Internet", Proc. of USENIX Symp. on Internet Technologies and Systems, 1997.

[17] G. Vossen and M. Weske, “The WASA2 Object-Oriented Workflow Management System”, Proc. ACM SIGMOD International Conference on Management of Data 1999,Philadelphia, PA, 1999, pp. 587-589.

[18] P. Koksal, I. Cingil, and A. Dogac, “A Component-based Workflow System with Dynamic Modifications”, Proc. of the Next Generation Information Technologies and Systems (NGITS'99), Israel, 1999.

[19] G. Cugola, 'Tolerating Deviations in Process Support Systems Via Flexible Enactment of Process Models”, IEEE Transactions on Software Engineering, Vol 24 Number 11, November 1998.