Building a Repository for Workflow Systems

Chengfei Liu
School of Computing Sciences
University of Technology, Sydney
NSW 2009, Australia
liu@socs.uts.edu.au

Xuemin Lin
School of Computer Science and Engineering
The University of New South Wales
NSW 2052, Australia
lxue@cse.unsw.edu.au

Xiaofang Zhou Maria Orlowska
Dept of Computer Science and Electrical Engineering
University of Queensland
QLD 4072, Australia
{*zxf, maria*}*@csee.uq.edu.au*

Abstract

Workflow technology is becoming the key technology for business process modeling, reengineering and automating. During the workflow specification, enactment and administration, there may appear various types of metadata about workflow specifications and instances. A repository manager is, therefore, necessary to store and manage these metadata. In this paper, a workflow metamodel is proposed first. Then the requirements of repository support for workflows are explored and analysed. Based on the requirements, an object oriented design for the repository manager is presented.

Keywords: *Workflows, Repository, Object-Oriented Design.*

1: Introduction

The requirement for streamlining business processes through re-engineering and automation has influenced research and development in workflow systems. Workflow management systems (WfMSs) [3] have been used for modelling, re-engineering and automating business processes to various degrees in application domains such as telecommunications, finance and accounting, manufacturing, office automation, and healthcare. The success of WfMSs has been driven by the need for businesses to stay technologically advanced in the ever-increasing competition of global markets.

A workflow is used to model and automate a business process by coordinating a set of tasks that are connected in order to achieve a common business goal. Typically, the tasks in a workflow systems are executed by different processing entities in a heterogeneous, distributed and autonomous environment. Each task defines a logical step that contributes towards the completion of a workflow, it may be completed by human, by an application system or by both of them. A workflow management system (WfMS) provides a set of tools for workflow model specification, workflow enactment, administration and monitoring of workflow instances.

During the process of workflow design, execution and administration, there may appear various types of metadata concerning the enterprise's business activities, organization and resources, the development, management and execution of workflows, etc. How to store and manage these data and provide services to WfMS tools falls into the matadata management, called repository system [1].

A *repository* is a shared database of information about engineered artifacts produced or used by an enterprise. A repository manager implements a layer of control services for modeling, retrieving, and managing the objects in a repository, usually on top of a DBMS. The repository technology is introduced in CASE/CAD environments to provide unified access and management to design data. Tools can work together by using common services supported by a repository management system, such as checkout/checkin, version and configuration control, notification, etc.

Currently, the market for repository systems is mainly driven by vendors of data dictionaries, CASE tools and CAD tools. There has been, to the best of our knowledge, no report on repository support for workflows so far in the literatures. Similar to a CASE repository [10, 6], there are many metadata management issues in a workflow repository, e.g., team development support for workflows, WfMS tools integration, dependency tracking and design changes propagation. Besides, it is also necessary for a workflow repository manager to support workflow execution and administration, e.g., services for scheduling, execution and tracking of a workflow, statistics report, resource discovery and management.

In this paper, we study repository support for workflows. The rest of the paper is organized as follows: In section 2, we present a metamodel for workflows. Section 3 discusses the requirements of repository for workflows. Based on these requirements, Section 4 introduces an object-oriented design of the workflow repository manager. Section 5 concludes the paper.

2: A Metamodel for Workflows

Business processes form a backbone of enterprises, they model routined work within an enterprise, describe how individual tasks (activities) inside an enterprise can be connected or coordinated to achieve a business process goal, and how processing resources and data resources can be used to perform each task. For example, a telecommunication company may have New Service Provisioning, Service Change Provisioning, Customer Billing processes etc. The New Service Provisioning process consists of tasks to collect accurate information from customers and create a corresponding service order record, then to provide line provioning either using existing facilities or physical installation of new facilities, and change the telephone directory, finally to update the telephone switch to activate service and then generate a bill. The tasks in the process may access data in some databases, such as customer database, facility database, directory database and billing database.

A metamodel of workflows is fundamental to study a workflow repository. The metamodel determines how information is stored and accessed in the repository, how well data integrity can be maintained, and how easily the existing model can be extended to accommodate new needs. Therefore we propose a workflow metamodel as shown in Figure 1.

A *workflow specification* consists of a set of weak entities *workflow task* as its components. A workflow task invokes a workflow specification. Workflow dependent task

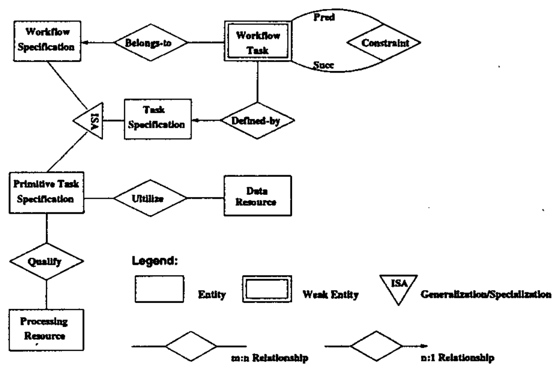

Figure 1. Conceptual Model of Workflows

properties are defined on a workflow task such as whether the task is critical to a workflow, etc. *Constraints* between tasks are treated as a unary relationship of workflow tasks. In specifying different types of constraints, we form a high level entity called *construct* which is shown in figure 2. Based on the relationship between two set of tasks: presecessor tasks and successor tasks, the construct is specialized to several subtypes which comply with the Workflow Management Coalition [3]: sequential, conditional, iterative, split(And/Or), and join(And/Or).

A *task specification* specifies workflow independent properties of a task, e.g., input/output parameters, visible states and compensatability of the task [9], etc. It may be invocated by more than one workflow tasks from the same or different workflows. For the purpose of nesting, a workflow specification is treated as a subtype of a task specification, it can be invoked as a workflow task in another workflow specification. Another subtype of a task specification is a *primitive task specification*. The properties of this entity set include implementation of the task (i.e. the task body), the processing resources and data resources used to execute the task, etc.

There are two appealing features in the metamodel compared with [2]: 1). We explicitly distinguish workflow dependent aspects of a task (modelled as *workflow task*) from workflow independent aspects of the task(modelled as *task specification*), this enables independent development and reuse of tasks. 2). By defining *workflow specification* and *primitive task specification* as subtypes of *task specification*, the model makes nesting and modularization of workflows possible.

3: Requirements of Workflow Repository

To facilitate integration of tools in a workflow management environment, a workflow manager should provide the following services:

- basic operations for all data stored in the workflow repository
- version management and configuration control - A version [4] is a semantically meaningful snapshot of an object at a point in its lifecycle. A version history of an object

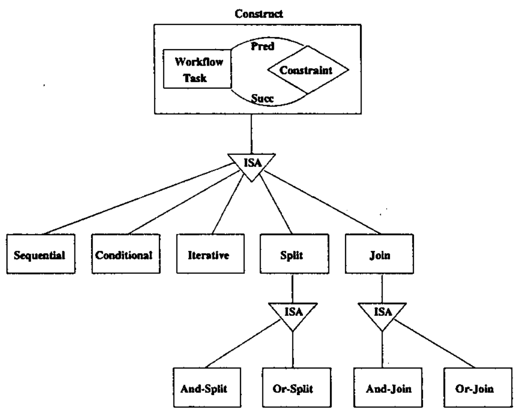

Figure 2. Subtypes of Constraints

can be represented as a directed acyclic graph. A configuration serves to group together objects that are to be treated as a unit for the purpose of versioning. The configuration control mechanism is used to make a binding between a version of a composite object and a version of each of its versioned components. Version management and configuration control is useful to support team development of workflows by allowing multiple designers working on different versions of same task specification or workflow specification. It is also useful to support evolution of workflow models [8].

- verification - Once a workflow specification is completed, a verification of its correctness is important. Verification services can be supported given some correctness criteria, e.g., acyclicness, reachability of a task, etc.
- structured browsing and editing of workflow structures
- worklist management - A worklist consists of a list of work items. Each work item is assigned to a processing resource during the run-time. A task instance may have one or more work items.
- run time tracking - It is useful to provide services for tracing state transitions of a workflow instance and record them into audit trail - a sequence of history records of the state transitions of a workflow instance from the start to completion.
- run time query facility - It is desirable to provide a query facility so that workflow users can monitor the execution states of a workflow or a task.
- performance turning - In order to improve workflow performance, a workflow system administrator may need statistic data from previous executions for analysis. For instance, we need to know average execution time of a workflow or a task, the coverage of a task in a workflow, the workload of the processing entities.
- resource management - The information about processing and data resources should be well organized and stored in the repository. Services for retrieving information about resources and finding proper resources might be supported.
- context management - To find resources easily, we can use a context dictionary to group resources. A context can be described by some properties and services.

4: Design of the Repository Manager

Objects in a workflow repository are complex. Storing complex objects in relational tables usually requires taking the objects apart and storing a representation in a set of flat tables. Reconstructing the objects involves a series of *selects* from the RDBMS and reassembling the objects. During this process, previous existing identity relationships must be preserved. In contrast, an OODBMS has rich facilities for complex object modelling. Complex objects are laid out in contiguous memory instead of splitting them into different tables. A relationship between objects only costs a pointer dereference. In addition, subtyping, version control and configuration management are usually supported by most OODBMSs. Therefore, we choose object-oriented approach for workflow repository management. In this section, we introduce an object-oriented design of a repository manager for enterprise workflows based on requirements.

4.1: Common Object Services

Every object in the repository may have some common properties, therefore, a class called *Object* is designed as the *root* of the class hierarchy. Besides, the class *ObjectSet* is designed to hold all instances of the class.

```
define class Object
  type tuple(name:             string,
             description:      string);
  new:                               Object,
  destroy(o: Object):                boolean.
define class ObjectSet
  type set(Object);
  createSet:                         ObjectSet,
  destroySet(os: ObjectSet):         boolean,
  add(os: ObjectSet, o: Object):     boolean,
  remove(os: ObjectSet, o: Object):  boolean.
```

4.2: Version Management and Configuration Control

The class *VersionedObject* defines a versioned object which consists of a set of *ObjectVersios. current* version and *versionCount* is recorded together with a set of methods, e.g., *checkout* and *checkin* a version, *notify* other versioned objects when a versioned object has been modified.

```
define class ObjectVersion
  type tuple(versionNumber: integer,
             predVersions:  set(ObjectVersion),
             succVersions:  set(ObjectVersion),
             object:        Object).
define class VersionedObject subtype of Object
  type tuple(versionSet:    set(ObjectVersion),
             current:       ObjectVersion,
             versionCount:  integer);
  checkout(vo: VersionedObject, vn: integer): ObjectVersion,
  checkin(vo: VersionedObject, pred: set(ObjectVersion), o: Object): integer,
  notify(vo: VersionedObject, others: set(VersionedObject)): void.
```

A versioned composite object is a natural configuration which groups together versioned component objects. When a version of a versioned composite object is checked out to a workspace, the corresponding versions of all its versioned component objects are also checked out to the space. If a change is to be made to the version of one of its versioned component objects, this component version needs to be checked out to a more private workspace from the workspace where its parent resides. After modification, one can checkin (back) the new version of the component to the workspace of its parent. New version of its parent composite object, therefore, may be created if one wants to checkin the modified parent object.

4.3: Workflow Design

The class *TaskSpec* corresponds to the entity *Task Specification* and the relationship *Defined-by* in Figure 1. Every task specification has a list of input/output parameters, reveals a set of visible states, has a set of properties, say, compensatable or not, may be referenced by tasks from several workflows. In order to make it easy to find when designing a workflow specification, we use *context* to categorize task specifications. The context management and task discovery issues will be discussed later.

```
define class TaskSpec subtype of Object
  type tuple(inputParameters:  set(Parameter),
             outputParameters: set(Parameter),
             visibleStates:    set(State),
             context:          set(Context),
             property:         set(Property),
             reference:        set(Task));
  addContext(ts: TaskSpec, c: Context):                boolean,
  removeContext(ts: TaskSpec, c: Context):             boolean,
  modifyContext(ts: TaskSpec, oc, nc: Context):        boolean,
  addProperty(ts: TaskSpec, p: Property):              boolean,
  removeProperty(ts: TaskSpec, p: Property):           boolean,
  modifyProperty(ts: TaskSpec, op, np: Property):      boolean,
  addReference(ts: TaskSpec, t: Task):                 boolean,
  removeReference(ts: TaskSpec, t: Task):              boolean,
  addInputParameter(ts: TaskSpec, p: Parameter):       boolean,
  removeInputParameter(ts: TaskSpec, p: Parameter):    boolean,
  addOutputParameter(ts: TaskSpec, p: Parameter):      boolean,
  removeOutputParameter(ts: TaskSpec, p: Parameter):   boolean.
```

As a subclass of TaskSpec, *PrimTaskSpec* records the task body of a primitive task specification, the processing and data resources used. A task or part of a task (work item) can be assigned statically or dynamically to a processing resource.

```
define class PrimTaskSpec subtype of TaskSpec
  type tuple(taskBody:       TaskBody,
             procResource:   set(ProcResource),
             dataResource:   set(DataResource));
  setTaskBody(ts: PrimTaskSpec, tb: TaskBody):            boolean,
  qualify(ts: PrimTaskSpec, p: ProcResource):             boolean,
  addProcResource(ts: PrimTaskSpec, p: ProcResource):     boolean,
  deleteProcResource(ts: PrimTaskSpec, p: ProcResource):  boolean,
  addDataResource(ts: PrimTaskSpec, p: ProcResource):     boolean,
  deleteDataResource(ts: PrimTaskSpec, p: DataResource):  boolean.
```

WFSpec is defined as another subclass of TaskSpec. It consists of a set of Workflow Tasks, with some tasks as initial, some as final. The operation *acyclicDirected* is used to check if the workflow model forms an acyclic directed graph.

```
define class WFSpec subtype of TaskSpec
```

```
  type tuple(taskSet:        set(Task),
             initialTasks:   set(Task),
             finalTasks:     set(Task));
  addTask(t: Task, w: WFSpec):               boolean,
  removeTask(t: Task, w: WFSpec):            boolean,
  addInitialTask(t: Task, w: WFSpec):        boolean,
  removeInitialTask(t: Task, w: WFSpec):     boolean,
  addFinalTask(t: Task, w: WFSpec):          boolean,
  removeFinalTask(t: Task, w: WFSpec):       boolean,
  acyclicDirected(w: Workflow):              boolean.
```

The class *Task* is designed to correspond to the entity *Workflow Task* in Figure 1. A workflow task acts as a component of a workflow specification. It defines how an independent task is participated in the workflow, including constraints between tasks (expressed by the attributes *preConstruct* and *postConstruct*), *property* (e.g.,, critical or not), *compensating task* defined for backward recovery [5, 9].

```
define class Task subtype of Object
  type tuple(workflow:       WFSpec,
             definedBy:      TaskSpec,
             preConstruct:   Construct,
             postConstruct:  Construct,
             property:       set(Property),
             compensation:   TaskSpec);
  createTask(w: WFSpec):                         Task,
  removeTask(t: Task):                           boolean,
  setWorkflow(t: Task, w: WFSpec):               boolean,
  setDefinedBy(t: Task, ts: TaskSpec):           boolean,
  setPreConstruct(t: Task, c: Construct):        boolean,
  setPostConstruct(t: Task, c: Construct):       boolean,
  addTrigger(t: Task, c: Construct):             boolean,
  removeTrigger(t: Task, c: Construct):          boolean,
  addProperty(t: Task, p: Property):             boolean,
  removeProperty(t: Task, p: Property):          boolean,
  setCompensation(t: Task, cp: TaskSpec):        boolean.
```

The class *construct* defines a constraint between two set of tasks: presecessor tasks and successor tasks.

```
define class Construct subtype of Object
  type tuple(predTask: set(Task),
             succTask: set(Task));
  addPredTask(t: Task, c: Constraint):        boolean,
  removePredTask(t: Task, c: Constraint):     boolean,
  addSuccTask(t: Task, c: Constraint):        boolean,
  removeSuccTask(t: Task, c: Constraint):     boolean;
```

As shown in Figure 2, a construct entity may be specialized to several subtypes: sequential, conditional, iterative, split, and join. These subtypes can be defined as subclasses of the construct class, e.g., the sequential, conditional and join subclasses can be defined as follows:

```
define class Sequential subtype of Construct
  type refine predTask with cardinality(predTask) = 1,
       refine succTask with cardinality(succTask) = 1.
define class Conditional subtype of Construct
  type tuple(condition: Condition);
       refine predTask with cardinality(predTask) = 1.
define class Join subtype of Construct
  type refine succTask with cardinality(succTask) = 1;
```

As designed above, repository objects together with their relationships constitute a labeled directed graph, with nodes representing objects and arcs representing relationships. A change made to an object, therefore, can be propagated along all dependency arcs originated from the objects [7]. How the change impacts on its related objects depands on the semantics of the dependency, i.e., the relationship between them and the type of the change operation. For examples, a *delete* operation on an object will delete all its *component* objects, but only unlink the references to its *referenced* objects.

4.4: Workflow Run-time Support and Administration

The class *WFInst* is designed to keep useful data about workflow instances. It is generated by scheduling tools, and used by tracking tools. It is also used by administration tools to generate statistics for the workflows. The attribute *wfSpec* is used to refer to the workflow specification from which the instance is made. The attribute *taskList* is used to record information about tasks which are really executed during the run-time. These include the processing resource assigned, the values of input/output parameters, etc. The attribute *log* is used as audit trail of the workflow instance. It keeps all state transition information of the instance. The state information of the workflow instance and its tasks can be derived from the audit trail.

```
define class WFInst subtype of Object
  type tuple(wfSpec:    WFSpec,
             taskList:  set(TaskInfo),
             log:       list(LogItem));
  workflowStatus(w: WFInst):         set(tuple(Task, State)),
  addTask(w: WFInst, t: TaskInfo):   boolean,
  addLog(w: WFInst, l: LogItem):     boolean,
  taskStatus(w: WFInst, t: Task):    State.
define type TaskInfo
  tuple(task:              Task,
        procEntity:        ProcResource,
        inputParameter:    set(Parameter),
        outputParameter:   set(parameter)).
define type LogItem
  tuple(task:        Task,
        date:        Date,
        time:        Time,
        transition:  String,
        oldState:    State,
        newState:    State).
```

The class *worklist* is used to record work items in a workflow environment, and their assignments to processing resources.

```
define class WorkList subtype of Object
  type tuple(procEntity:     ProcResource,
             workItem:       list(WorkItem));
  addWorkItem(t: TaskItem, w: WorkList):    boolean,
  removeWorkItem(t: TaskItem, w: Worklist): boolean,
  resourceStatus(d: Date, t: Time):         string.
define type WorkItem
  tuple(date:       Date,
        time:       Time,
        duration:   Time,
        wfInst:     WFInst,
        task:       Task).
```

4.5: Resource Management

The class *ProcResource* is designed to correspond to the processing resource entity. A processing resource, from the perspective of practical use in workflows, can be abstracted as providing a set of services with certain invocation formats. Each service has a signature, belongs to some contexts, and has a set of properties. *ProcResource* can be specialized to several subclasses: *Application*, *Surrogate* and *Human*. An application system can be a customized application programs, or a general purposed software system. A Surrogate is designed to hold useful information of a foreign resource that tasks may invoke. A task may also be accomplished by human being.

```
define class ProcResource subtype of Object
  type tuple(function:          set(Service),
           invocationFormat:  string).
define type Service:
  tuple(name:              string,
        inputParameter:    set(Parameter),
        outputParameters:  set(Parameter),
        context:           set(Context),
        property:          set(Property)).
define class Application subtype of ProcResource
  type tuple(executable:  File,
           sourceCode:  File,
           document:    File,
           supportedBy: Application,
           mountedOn:   Machine).
define class Surrogate subtype of ProcResource
  type tuple(location:    Location,
           ownership:   Organization,
           authority:   Authority).
define class Human subtype of ProcResource
  type tuple(title:       string,
           address:     string).
```

A data resource is modelled by the class *DataResource*.

```
define class DataResource
  type tuple(location:    string,
           ownership:   string,
           executedOn:  Application,
           accessMode:  integer,
           security:    AccessControl,
           structure:   string).
```

The class *Context* is used to classify resources in terms of characteristics. A context forms a lattice.

```
define class Context subtype of Object
  type tuple(super:      set(Context),
           alias:      set(string),
           properties: set(Property));
  addSuper(sc: Context):                boolean,
  removeSuper(sc: Context):             boolean,
  addAlias(a: string):                  boolean,
  removeAlias(a: string):               boolean,
  addProperties(p: Property):           boolean,
  removeProperties(p: Property):        boolean.
define class ContextSet subtype of ObjectSet
  type set(Context);
  showContextLattice(c: Context):       set(Context),
  listContext(c: Context, p: Property): set(Context).
```

In the class *TaskSpec* and the class *ProcResource*, the attribute *context* is included for the purpose of finding proper tasks or processing resources using context. For example, search methods based on context can be added to the class *TaskSpecSet* to find appropriate tasks.

```
define class TaskSpecSet subtype of ObjectSet
  type set(TaskSpec);
  listTasks(c: Context):                      set(TaskSpec),
  search(c: Context, ps: set(Property)):      set(TaskSpec),
  bestMatch(c: Context, ps: set(Property)):   TaskSpec.
```

5: Conclusion

In this paper, we proposed a metamodel and explored the requirements to build a repository for workflow systems. An object-oriented design of a repository manager was presented. Currently, a preliminary prototype of a workflow repository manager based on the above design has been developed using ObjectStore OODBMS in SmallTalk VisualWork2.0 environment. The implementation turns out that developing a workflow repository manager based on an OODBMS is quite productive as some services such as version management can be easily implemented using functions provided by ObjectStore.

References

[1] Philip A. Bernstein and Umeshwar Dayal. An overview of repository technology. In *Proceedings of the 20th VLDB Conference*, pages 705–712, 1994.

[2] F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Conceptual modeling of workflows. In *Proceedings of OO-ER conference*, pages 341–354, 1995.

[3] Workflow Management Coalition. *The Workflow Management Coalition Specification - Terminology and Glossary*, 1996.

[4] R. H. Katz. Toward a unified framework for version modeling in engineering databases. *ACM Computing Surveys*, 22(4):375–408, December 1990.

[5] D. Kuo, M. Lawley, C. Liu, and M. Orlowska. A general model for nested transactional workflows. In *Proceedings of the International Workshop on Advanced Transaction Models and Architectures*, pages 18–35, Goa, India, September 1996.

[6] H. C. Leftkovits. *IBM's Repository Manager/MVS: Concepts, Facilities, and Capabilities.* QED Technical Publishing Group, 1991.

[7] C. Liu, H. Li, and M. Orlowska. Supporting update propagation in object-oriented databases. *Data and Knowledge Engineering*, 26(1):99–115, 1998.

[8] C. Liu, M. Orlowska, and H. Li. Automating handover in dynamic workflow environments. In *Proceedings of Advanced Information Systems Engineering 10th International Conference (CAiSE*98)*, pages 159–171, Pisa, Italy, June 1998.

[9] C. Liu, M. Orlowska, X. Zhou, and X. Lin. Confirmation: a solution to non-compensatability problem in workflow systems. In *Proceedings of the 15th International Conference on Data Engineering*, Sydney, Australia, March 1999.

[10] L. Wakeman and J. Jowett. *PCTE - The Standard for Open Repositories.* Prentice-Hall, 1993.