Code Versioning in a Workflow Management System

by

P. Alexander Rolfe

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2002

Author . . . .
Department of Electrical Engineering and Computer Science
May 24, 2002

Certified by . . . .
Michael D. Ernst
Assistant Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Code Versioning in a Workflow Management System

by

P. Alexander Rolfe

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 2002, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

Workflow systems implement process definitions in laboratory, office, and industrial settings. In many cases, the process definition is implicit in the ad-hoc software written for a particular task. In other cases, a generic framework provides basic functionality and structure, offering quicker development and advanced features. Few workflow systems handle changes in the process definition or the implementing code. This work shows that complicated workflow processes can be composed of a few simple primitives and that changes in the workflow structure and code can be managed effectively.

Thesis Supervisor: Michael D. Ernst
Title: Assistant Professor

Acknowledgments

My thanks goes to Michael Ernst for supervising this thesis and to Brian Gilman and the Whitehead Institute/MIT Center for Genome Research for supporting the work. I also owe much of my inspiration and insight into the problems of workflow systems to the LIMS group at the Genome Center: Brian Gilman, Erich Stahl, Will Fitzhugh, Ralph Santos, Andrew Zimmer, Daniel Richter, and Catherine Hosage Norton, and Jay Ireland.

Contents

1 Introduction **13**

1.1 Why Are Workflow Systems Useful? 14

1.2 Definitions 15

1.3 Workflow Change 16

1.4 Related Systems and Other Work 17

1.4.1 Workflow Management Coalition 18

1.4.2 LabBase and LabFlow 19

1.4.3 MicroWorkflow 20

1.4.4 Dynamic Workflow Modification 20

1.4.5 Ad Hoc Workflow 21

2 Design of a Workflow System **23**

2.1 Assumptions 23

2.2 Requirements 24

2.3 System Components 27

2.3.1 Workflow Description 28

2.3.2 Translation 30

2.3.3 User Interface 30

2.3.4 User Code 30

2.3.5 Workflow Engine 31

2.3.6 Workflow Storage 33

3 System Implementation **37**

3.1 Workflow Description Classes 37

3.2 Base Runtime classes 40

3.2.1 Materials 40

3.2.2 Events 40

3.2.3 `Storable` 41

3.2.4 Listeners 42

3.3 Workflow Storage Component 44

3.3.1 Process Definition 44

3.3.2 Process Instance Status 44

3.3.3 Process History 45

3.3.4 RDBMS Dependencies 45

3.4 Runtime Engine 46

3.5 Workflow Design Tool 49

3.6 Implementing Translation 51

3.6.1 Sub-Workflow 51

3.6.2 Workflow Chaining 54

3.6.3 Blocking and Scheduling 54

3.6.4 Non-Determinism 54

4 Versioning **57**

4.1 Types of Changes 57

4.1.1 Structural Change 58

4.1.2 Parameter Change 59

4.1.3 Code Change 59

4.1.4 Global or Limited Changes 59

4.2 Design 60

4.3 Implementation 61

4.3.1 Rewriting 62

4.3.2 Workflow Deploy 64

4.3.3 Workflow Engine and Storage 65
4.4 Other Approaches to Versioning 65

5 Potential Features 67
5.1 Constraint Checking 67
5.2 Data Flow Analysis 70
5.3 Interfaces to Other Languages 70

6 Conclusion 71
6.1 Evaluation 71
6.2 Contributions 72

A Workflow Storage Schema 73

B System Performance 77
B.1 Dispatching Events to the Initial State 78
B.2 External Program 79

List of Figures

1-1 A Sample Workflow 17

2-1 An Example High Level Workflow Description 25

2-2 Major System Components 35

3-1 Runtime Dataflow 38

3-2 Low level workflow description classes 39

3-3 JMS Implementation: Layers of Listeners 46

3-4 Process Design Tool Screenshot 50

3-5 Implementing Blocking or Scheduling 55

3-6 Using Extra States to Model Nondeterminism 56

4-1 An Example of Structural Change to a Workflow Process 58

4-2 Mapping Class Name and Version to New Name 63

5-1 Constraint Checking as a Translated Feature 69

B-1 Sample Workflow for Performance Testing 78

Chapter 1

Introduction

Workflow Management Systems enact process definitions, manage the resulting process instances, and report on the results. Unlike traditional software processes, the Workflow process often involves both computational work and physical work that exists outside the computer system. Steps in the process generally involve a single, well defined input and no global state. For example, a bank considers each loan application independently using a well defined process and might use a Workflow System to organize the activities and data related to loan approval. While the same process could be implemented by custom software written for a particular task, a Workflow Management System provides a framework and pre-built functionality. The Workflow Management System described in this work provides a modular system that can implement most process definitions and conveniently handles changes to those definitions.

Our work makes two contributions to the study of Workflow Management Systems by addressing extensibility and change. We describe the design and implementation of our workflow system that allows complicated functionality to be created with a minimal set of primitives. Our system allows users to employ their own implementation or reuse a previous implementation of advanced features if the default implementation is unsuitable. Our system also encourages programmers to write reusable code that may be used with multiple processes. Furthermore, our implementation remains simple even as users add features because the features are built on top of the system rather than integrated into the core. Many other systems provide complex primitives for scheduling, authorization, roles, worklists, containers, and domain-specific features. In doing so, they frequently prevent extensibility, lock the user into the provided feature implementation, and make the core system difficult to maintain.

While many Workflow Management Systems model processes, few model the changes that may occur in a process. Our second contribution is to discuss the design and implementation of extensions to handle revisions to a Workflow process. In particular, we address the problem of running multiple versions of a process simultaneously.

Before considering Workflow Management Systems further, we present a short case for their utility. Then we define the terms that we will use in our discussions and introduce the problem of change in workflow systems. Our introduction to workflows concludes with a brief survey of other Workflow Management Systems.

The second chapter of this work describes the design requirements and design decisions made for the implementation of our Workflow Management System. Chapter three describes our implementation. Chapter four describes modifications to our system to support change in the workflow schema or code. Chapter five discusses several potential features that we have not yet implemented.

1.1 Why Are Workflow Systems Useful?

Consider a high throughput scientific research laboratory such as the Whitehead Genome Center where a researcher has asked a programmer for support in a project that involves laboratory and computational work to perform thousands or millions of experiments per week. The programmer must produce a software system that can take input from the researcher, create worklists for the laboratory technicians, accept laboratory results, record data, perform analyses of the data, and provide statistics and reports about the throughput of the whole project.

Projects often start on a small scale. Sometimes the researcher abandons the project because of poor results. In other cases, the experiments work well and the project is scaled up dramatically. Without suitable tools, the programmer must write the system from scratch, requiring decisions about how much time to spend building software infrastructure. Since she cannot predict which course a project will follow, a developer cannot justify a large time investment to create custom software to manage the data for the project. For example, writing a comprehensive reporting suite that can track data through whatever ad-hoc pipeline the programmer has created will take a long time. Even if the programmer were willing to devote the necessary time, the researcher probably doesn't want to wait months or years to start their work.

Without tools, the programmer may repeat errors made by implementors of previous systems [6, 7]. Experience shows that systems built without the appropriate tools tend to be hastily written, fragile, and unable to scale if the project is ramped up. For example, many systems fail to handle unusual input correctly or suffer data corruption when programs run simultaneously or crash. This situation calls for a set of tools to provide a framework for developing workflow systems by abstracting out the common features, allowing the programmer to concentrate on the unique aspects of the task at hand, and promoting reuse of code between projects.

1.2 Definitions

Workflow Management System software tracks data from, analyzes data from, reports on, and drives processes. In general, these processes involve physical steps such as laboratory experiments, document processing, transportation, or manufacturing, in addition to computational steps. A **Workflow Process** is specific process definition of work that is to be performed. A **Workflow Instance** is an instantiation of a Workflow Process on one Material. An Instance takes exactly one of the possible paths through the Process. A series of **Steps** processes computation or physical resources called **Materials**. A Material in the system can represent a physical object such as a blood sample at a hospital diagnostic laboratory or it can represent a computational object such as the alignment between two biological sequences. An **Event** object records the outcome of a Step and contains a **Result**, a string summarizing the

Event. For example, a Step might check that a gel from a DNA sequencing machine has an average quality greater than 20; the Result might be OK if the average quality is greater than 20 or FAIL otherwise.

We label Materials with **States** to describe their current status in the Workflow Process. A collection of Materials in the same State forms a queue for processing by a specific Step. The State of a Material will change as the Material moves through the process and the system places the Material in different queues. The set of Steps (recorded as Events and their Results) performed on a Material comprises the Material's **History**; a Material's History uniquely determines its current State in the workflow [2, 1].

In our system, classes called **Listeners** perform the Steps. Each State in the process may contain Listeners. When a Material arrives in that State, the system invokes the Listeners on the Material. The Listener code is not part of the core workflow system; it is generally provided by the users to implement process-specific work.

A Workflow Process, often referred to as a workflow, resembles a state machine or graph. A Material moves through the workflow, moving from State to State along the edges of the graph. Each transition corresponds to some Step that occurs on the Material. The set of edges traversed is the History of the Material. Figure 1-1 shows a small example process.

1.3 Workflow Change

Workflows rarely model static processes. The users of the Workflow System will want to change the process structure and code to reflect the new physical process. In ideal circumstances, the new process and code apply to all Materials in the system. Unfortunately, circumstances sometimes prevent this type of change; some Materials may need to continue in the old process while others follow the new process.

A Workflow Management System should provide a language or method for describing changes to a workflow process's structure, a language or method for describing

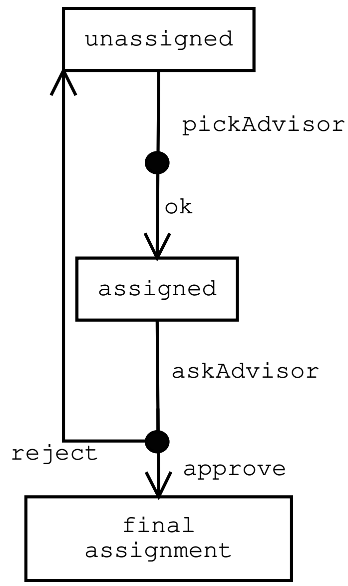

Figure 1-1: **A Sample Workflow** This process contains three States (the boxes) and two Steps (the lines leaving the boxes) to process Materials representing a student's choice of advisor. New choices begin in the `unassigned` State and are processed by the `pickAdvisor` Step. This Step has only one valid result, `ok`. Choices which have been processed by pickAdvisor are placed in the `assigned` State and then processed by the `askAdvisor` Step in which the chosen advisor may review the decision. The advisor may `approve` or `reject` the decision. The system moves choices with the `approve` Result to the `final assignment` State. Choices with the `reject` Result return to the unassigned State.
If the advisor approves a student's choice, the History of the choices Material will include a `pickAdvisor` Step and an `askAdvisor` Step.

migration for in-progress processes from an old process definition to a new process definition, and the ability to effectively manage and run multiple versions of a process. Shazia Sadiq's papers on workflow schema changes address the first two issues by proposing such languages [8, 9, 10]. However, no Workflow System besides our system adequately addresses changes to the code.

1.4 Related Systems and Other Work

While we have provided a terminology and will discuss a design and implementation for our Workflow Management System, many other designs and implementations are possible. Other systems have emphasized different features to accommodate different ideas of exactly what a Workflow Management System is and what it should provide. We survey several other Workflow Management Systems to show what the range of desired features might be and to present designs that have influenced ours.

1.4.1 Workflow Management Coalition

The Workflow Management Coalition (WFMC) standards describe workflow process specifications, workflow enactment, and communications between workflow systems. The standards focus on business processes and provide features for work lists (assigning an open task to a specific person, machine, or group thereof) and interfaces for process control [4]. While few, if any, products fully implement the WFMC specification, much of the terminology is standard.

In the WFMC model, an administrator uses a definition tool to write the process definition. The workflow management engine combines the process definition and supporting information such as organization/role data to enact the process. The engine may "activate applications necessary to execute particular activities" and will place items into worklists when user interaction is necessary. The activated applications closely resemble our system's Listeners. Worklists are an excellent example of domain-specific functionality being added to a workflow system. Worklists are not essential to process enactment and are useful only when the system needs to assign work to specific users.

The WFMC's focus on business processes led it to include a variety of process control and supervisory functions in the engine-client API. For example, the API defines functions to stop, start and suspend process instances. The administrative API defines functions to assign roles, grant and revoke privileges, and allocate resources. These functions also reflect business-specific concepts that do not exist in many workflows that focus on computation.

Our Workflow Management System, and many written by others, share much of the terminology and general system structure of the Workflow Management Coalition. For example, most Workflow Management Systems include components for process design, process enactment, and process definition storage. Our system does not implement the WFMC specifications.

1.4.2 LabBase and LabFlow

Steve Rozen, Nathan Goodman, and Alex Smith developed LabBase and LabFlow at the Whitehead Institute/MIT Center for Genome Research and the Jackson Laboratory. LabBase provides Perl object storage, State recording, and History recording [2]. LabFlow provides a simple engine to implement a specified workflow process [1].

LabBase's object storage is based on a simple Object-Relational mapping defined in a schema that specifies the database types of the object fields. The LabBase libraries extract the relevant fields from Perl data structures and perform the necessary data interaction.

LabBase also provides State and History recording. In LabBase's terminology, objects can be labeled as Materials or Steps. Labbase can associate Materials with predefined tags known as States that describe the status of the Material in some process. LabBase also allows a change in a Material's State to be associated with a Step object. The ordered collection of Steps associated with a Material forms its History. Our concepts of State and History closely resemble LabBase's. While LabBase provides a mechanism to store State and History information, it has no concept of process definition to ensure that only valid States are assigned to Materials.

LabBase also provides the interesting feature that fields of Steps can be viewed as fields of associated Materials. This allows Materials to be declared with a minimal set of fields. Additional virtual fields can be accumulated from the Steps performed over the course of the process.

LabFlow provides a simple process engine using a user-provided process schema (definition) and user-provided classes that implement the actual work. The user can specify a series of steps to be performed and provide an implementation for the steps by provided code that implements an appropriate interface. LabFlow does not provide a good method to handle asynchronous Steps (e.g., those that require waiting for user input). LabFlow also lacks a system, similar our high level to low level translation, for encapsulating common behaviors.

1.4.3 MicroWorkflow

Dragos Manolescu's MicroWorkflow paper criticized the size and feature-creep of many commercial workflow systems [5]. In particular, Manolescu claimed that systems limited flexibility by not providing a means to use custom components, instead providing many default features. The MicroWorkflow system takes an alternate approach; a few simple components provide core workflow functionality and allow the user (or programmer) to use whatever components are desired.

The components of Manolescu's system are similar to ours and to the WFMC's: an execution core, a monitoring component, history recording, persistence, worklists, and workflow combination. We also share the goal of producing a modular and extensible system.

1.4.4 Dynamic Workflow Modification

Shazia Sadiq addresses the problems of handling in-progress process instances when the underlying process definition changes [10, 9, 8]. Sadiq defines five "modification policies" to describe workflow change: flush, abort, migrate, adapt, and build.

The flush policy allows active instances to complete the old process definition but all new instances follow the new process. Abort is the simplest modification policy; the system stops active process instances and remove them. The migrate policy moves process instances from the old process to the new. Migration can be complicated if certain paths through the process become invalid; the policy must define rules to modify any valid process instance such that it is valid in the new process definition. The adapt policy handles cases in which the process definition changes temporarily (e.g. exception handling). The build policy includes all changes in which the new process is built from scratch rather than being based on an existing process.

The Sadiq papers define a method for implementing these policies based on classifying instances and on Compliance Graphs that map instances from the old process definition to the new. This technique complements our approach to managing process change. Sadiq describes a method for migrating processes but does not discuss any techniques to simultaneously run two processes, probably because the migration technique addresses Workflow Management Systems in general rather than a particular system.

1.4.5 Ad Hoc Workflow

In some settings, the process definition cannot be accurately or reliably specified before the process begins. The process may be too complicated, as is often the case when humans are required to make decisions about how to handle Materials. In other cases, the process is not well understood. Whatever the cause, these settings would render a process definition useless because they would generate a huge number of exceptional cases. Here, the best that workflow systems can do is to record the work and attempt to report after the fact. Our design allows ad-hoc movement through a process only in exceptional circumstances. In general, we require Materials to follow the process specification.

Chapter 2

Design of a Workflow System

We now turn to the design requirements for our workflow system by discussing our assumptions and enumerating the system's user-visible functionality. These assumptions and requirements lead to the design of the system components and their interactions.

2.1 Assumptions

We assume that a Material can reside in at most one State in a workflow. We also assume that the State uniquely defines the work that is to be performed; however, in some cases, exceptional events occur, and the workflow system should handle them gracefully. Finally, we assume that a process definition has exactly one initial State.

In some physical systems, a Material may be in more than one State, allowing a process to branch and then merge again later. For example, two quality checks may be performed on a sample in parallel to reduce the latency of the laboratory process. Our system does not allow two parallel actions in a single workflow process. Instead, we model our workflow on a state machine and allow a Material to have at most one State in a process at a given time. Allowing only a single State simplifies both the implementation of the system and reasoning about the system; there can never be any ambiguity about a Material's location in the process. In other words, a Material's State uniquely identifies the next Step; we do not allow a situation where one of several Steps might process some Material.

We require that only the transitions specified in the Workflow description be allowed under normal circumstances, forbidding ad-hoc workflows. In an ideal setting, we would be able to model every process in advance and with perfect accuracy. Experience indicates otherwise; in some cases, a transient system failure or some other exceptional circumstance will require that we violate the structure of the workflow and manually move a Material from one State to another. The system must allow a user to force such transitions.

We require that a Workflow process have a single initial State, providing each Workflow with a unique starting point. A Workflow may have multiple terminal States. The single-beginning, multiple-ending characteristic reflects the fact that a process should begin at a single Step but may branch along several paths as the work proceeds.

2.2 Requirements

The Workflow system should allow the programmer to concentrate on the high-level modeling of the laboratory process. Furthermore, it should encourage code reuse. These goals can be accomplished by providing an intuitive, high level process description language that makes using existing components and combining existing workflow easy. The interfaces that user-components implement should encourage reuse by providing minimal context to the component and by allowing per-workflow parameterization.

Workflow Design and Representation

The system should provide an intuitive way to describe the process of interest. The most natural way to model a process is with something resembling a graph. Nodes in the graph represent States in the process and edges represent transitions between States (i.e. Steps). Workflow development begins with the specification of the workflow process: enumeration of the States and the connecting Steps. This information is data provided to the workflow system [3]. Figure 2-1 shows an example of the high level graphical language.

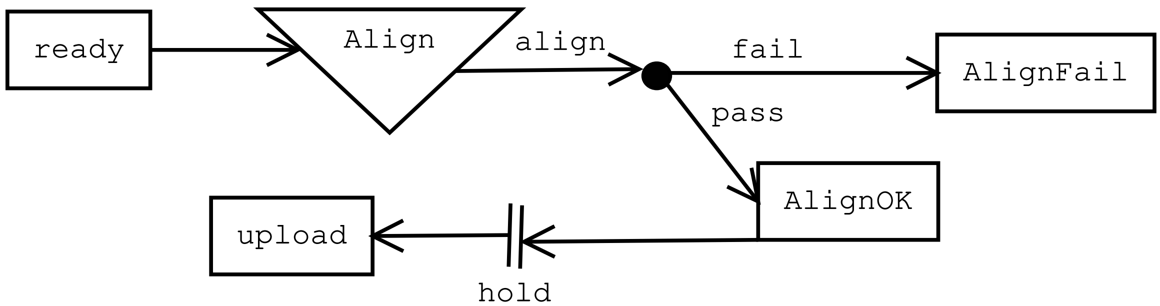

Figure 2-1: **An Example High Level Workflow Description** This sample workflow process definition uses a sub-workflow and scheduling. The square boxes represent simple States. The triangle represents a call to another workflow. The entire sub-workflow is executed as a single Step and its terminal State determines the Result of that Step. If the alignment process has failed, the Material is moved to the AlignFail State. The parallel lines indicate that Materials will be held until some scheduling criteria is met.

User Code Interface

To enact the workflow, the developer must provide code to implement the specific functionality of the system. This code implements (in the case of computational steps) or represents (in the case of physical work) the actions of the process. Our system must provide a simple interface for user code and must encourage the development of reusable code.

The system combines the workflow process description and the provided code into the implementation of the workflow process. For each transition in the workflow graph, the system calls the appropriate user code for the step or accepts input from an external program. Having performed this step, the system adds the resulting Event to the Material's History, moves the Material to the correct State, and then proceeds to the next step in the process.

The interface between the system and the user code should be simple. The user code for Listeners receives the input Material and the previous Event in the process but does not know its position in the process or the next step. In our design, the Material moving through the system is the basic unit of work and information. Consequently, each Step should require only that Material as input. The code for the Step should not require access to global state.

We do not allow the user code to specify the destination State for the Material; instead, we allow the code to return a Result that determines the Material's routing. If we allowed the Listener to specify the next State, then all of the Listeners in a process would be tied to each other, making reuse without modification difficult. For example, if class A referred to B which referred to C and we replaced C with Z, then we would need to replace or modify B and therefore A as well. By making the system map Results to States, we add a layer of indirection that allows and encourages user code to be more general than the process using it.

External programs that communicate with the system also need a simple interface for dispatching Events into the system. External programs need to dispatch Events to a particular State (generally the initial State of a process) and dispatch Events such that the appear to come from a particular State.

Workflow Parameterization

Workflows sometimes share the same structure (States and Event results) and code but differ in details. For example, a Step that checks the result of a laboratory test might need a higher quality threshold in one workflow than in another. Workflow Parameterization allows the developer to specify runtime parameters for the user code.

Reporting

The Workflow system should also provide facilities for report generation. With its standardized process representation and History recording, the system can generate many simple reports about the progress of Materials through the workflow [6]. For example, the system can report how many Materials were processed by a Step during a specified time interval. Our design does not yet include a reporting system.

Workflow Revisions

Workflows often change, either in the structure of the process being implemented or in the details of the Steps performed. A successful Workflow system needs to handle changes or revisions in the workflow in a convenient manner [7]. Chapter 4 discusses Workflow revisions in detail.

2.3 System Components

We now discuss the major components of our design.

- A language to describe Workflows at a high level.
- A language to describe Workflows at a low level.
- A tool to translate the high level language into the low level language.
- A tool to design high level workflow descriptions.
- Classes to represent the Materials. To the Workflow system, these classes are basically records (simple information containers).
- Classes to implement the work (Listeners). These classes take in a Material and perform the appropriate work. Some of these will be single classes, while others will be external programs (not Listeners and not called by Listeners) that feed results to the system.
- Classes to record the work results (Events). These too are simple records produced by the work classes and associated with Materials to form the Materials' History.
- A Workflow Engine that accepts a workflow description and the user code and enacts the process.
- A Storage Engine. The storage engine must store both the description of the process, the process's run-time state, and serialized versions of the Materials and Events.

2.3.1 Workflow Description

Our Workflow Management System uses two levels of process description. At the high level, a GUI that presents the workflow as a graph to show how Steps fit together. However, a high level description might be a complicated entity with sub-workflows, parallel processes, dependencies, scheduling, and chaining. Such complexity would be difficult to handle in the actual Workflow Engine because of the variety and scope of features needed; the number of potential interactions between features might make testing the Engine difficult [5]. Furthermore, adding a new feature would require modifying the core engine rather than a separate, peripheral component. Consequently, we have decided to use two levels of workflow description. The designer and users of the system use the GUI to create a high level description that can include all of the previously mentioned features. The high level workflow is translated into a lower level workflow description that is simple to implement.

Translation promotes extensibility by providing a well defined means to add features to the system without knowledge of the system's internals or modifications to the existing code. Furthermore, by implementing each feature as a separate set of filters, we provide clear boundaries between each feature and the core engine. These boundaries encourage modular implementations of the features. In particular, we want to avoid a complicated engine implementation in which the various features depend on each other.

High Level

The simplest workflow is a linear progression of Steps. Our high level description language includes a variety of features that can add complexity:

- Branches. Materials come to a fork in the process and can follow one route or another. In most cases, the branch selection algorithm is deterministic and based on the History of the Material. A simple case of Branching routes the Material to Step $x+1$ based only the Result of Step x (this is the default method of routing Materials from one State to another). In figure 2-1, there is a branch after the Align sub-workflow based on the sub-workflow Result.

- Joins. Two paths through the system merge.
- Parallelization. Materials come to a fork in the process and proceed along both paths in parallel. The paths could be paths through the same workflow or through sub workflows.
- Synchronization. Synchronization can take several forms. First, it can operate as the inverse of parallel execution. Two paths join and a Material may not proceed past the synchronization point until it has completed both paths. In the second form, a Material might need to progress through two workflows A and B in parallel and the execution of the Steps of A and the Steps of B must be coordinated. For example, the third Step of B must happen at the same times as the fourth Step of A.
- Sub-Workflow. A Step in one workflow is another workflow.
- Chaining. Materials move through workflow A and then through workflow B (but do not return to A, as they would if B were a sub-workflow).
- Timing and Scheduling. A workflow might have timing constraints that prevent Materials from moving too fast, send notices if they move too slowly, or process Materials on some schedule. For example, a certain computational Step should only be run at night when the relevant machines are unused by interactive jobs.

Low Level

Our low level description does not directly implement most of the features available in the high level description. Instead, we use a very simple representation of the process. The description consists of a set of States and a set of Transitions. Each State corresponds to one set of actions. Each State contains one Transition object that maps the Result of the State's actions to the next State.

Figure 1-1 shows a graphical representation of a low level description.

2.3.2 Translation

Since the low level description object cannot represent all of the desired high level features, the user interface translates the high level description into a low level workflow description. System-provided Listeners implement various features in the high level description. To the Workflow Engine, these Listeners are no different than the Listeners provided by the developer. However, the translation system provides them to implement the high level features. For example, chained workflows can be implemented as two separate workflows. The translator adds a Listener to the terminal State of the first workflow that puts the Materials it sees into the second workflow.

2.3.3 User Interface

The user interface resembles a graph drawing program to allow the user to specify the structure and parameters of a process. When the user has specified the process structure, the UI deposits a WorkflowDescription object into the persistent storage using the Storage component (section 2.3.6). The GUI accepts dynamically provided pluggable widgets for use in workflow design. Each such widget represents one Step that can be added to a workflow and consists of :

- A factory to produce whatever objects represent States in the UI's representation of the workflow process.
- A filter that maps an instance of the workflow process representation to another instance of the workflow process representation, replacing high-level features with the low-level equivalent.
- Optionally, icons or renderers for use in graphical interfaces.

2.3.4 User Code

Users provide three types of code. Listeners implement the actions in the workflow. Listeners must provide a method that accepts an Event and a Material and performs the action. Our design does not force Events and Materials to provide any methods to the system as these objects are records for the process-specific data required by and produced by the Listeners.

2.3.5 Workflow Engine

The Workflow Engine provides run-time support for the process. When executed, the engine first obtains a workflow description. Our implementation accepts command line parameters to specify the name and version of a workflow that is to be retrieved using the Workflow Storage component (section 2.3.6). The engine then instantiate Listener objects for the States or some subset thereof. One might wish to instantiate only a subset of Listeners such that particular Steps are run on a particular machine or to better distribute the load from computationally intensive Steps.

Having completed its initialization, the engine accepts Events from Listeners and external programs, passing those Events to the appropriate Listeners. While passing Events, the engine updates the State and History of Materials (using the Workflow Storage).

Public Interface

To the developer, the Workflow Engine is relatively simple. It can be configured by providing a low-level description of the process and Listeners. Our implementation accepts command line paramenters to specify a process name and the information needed to connect to SQL and JMS servers (see chapter 3 for more information). We do not provide a Java interface that can invoke the engine.

Our design specifies a minimal Java interface to the Workflow Engine for dispatching Materials to the Engine. This interfaces provides two methods:

1. Dispatch an Event to a State : allows an external program to send an Event to a particular State for processing. Only programs that dispatch Materials to the initial State of a Workflow should use this feature. External programs should generally not dispatch Events directly to the non-initial States of a workflow. For an external program to know which to which State an Event should be

dispatched, the program would need to know the workflow structure. While this is possible (as we provide a means to retrieve workflow definitions), most programs should leave Material routing to the system to ensure that the routing is performed correctly (in case other work is done in conjunction with routing).

2. Dispatch an Event from a State : allows an external program to send an Event that is to be dispatched as if it were from the specified State. This can be used with any non-terminal State in the system. It will be used primarily by programs that process Materials in States that do not create a new Event.

Failure Characteristics

Any engine implementation must have well defined failure and reliability characteristics. When running, the engine passes Events from one State to another. Clearly any implementation that drops Events is unacceptable as users will be unhappy when workflow instances disappear because Events were lost. If the Engine software crashes or is halted by a system or network failure, we need to know what happened to any active Events. It may be acceptable in this situation to simply inform a user that there was an error with the messages and require handling outside the system. We consider this reliable as long as all such cases are reported. In general, however, we expect that the system must reliably and properly deliver all messages.

In addition to handling failures of message delivery, the engine software must be able to handle Listener failures. If a Listener fails by throwing an Exception, then the system has several options:

- Ignore the problem. If an active Listener fails, then the system loses track of the Material.

- Log the error but otherwise ignore the problem. The users of the system are responsible for cleaning up any problems.

- Retry the Listener. While this might be useful for transient errors, there is no guarantee that a Listener can be rerun safely.

- Stop. Stop processing the Material and notify the users or administrator.

Our system implements the last behavior. If a Listener fails, the system stops processing the Material and puts it aside for later handling by the system's users or administrator. We considered using the Retry behavior in some circumstances. If the designer of a workflow process could mark Listeners as idempotent, then the system could automatically retry those Steps up to some maximum number of times. We chose not to implement this behavior as it adds complexity but does not completely solve the problem as not all Listeners can be safely retried.

2.3.6 Workflow Storage

The first function of the workflow storage is to persistently store Workflow description objects. The second function is to store the run-time state of the system, namely the State and History of Materials. Finally, the storage component allows access to the current State and History of the Materials.

Splitting the Workflow Storage from the Workflow Engine accomplishes several goals. First, it accurately reflects the purposes of the two components. It seems slightly unnatural that the Workflow Engine, a run-time construct, should be queried for Material History or process descriptions. More importantly, it separates the implementation of the two components, allowing one to be replaced or modified independently of the other.

The Workflow Storage component records Material History and State. A Material is in zero or one States in any workflow. When a Material moves to a new State, the system removes the Material from the old State. When the Workflow Engine uses the Workflow Storage, the standard pattern is to move the Material from one State to another while adding an entry to the History; the Workflow Storage component should support this as a single atomic operation. History is an append-only list of the Events that have been associated with the Material[1].

[1]Ideally, it is append only. In reality, entries in the History may be deleted or modified to correct software or laboratory errors.

State and History Reporting

The Workflow Engine is the primary writer to the Workflow Storage. However, any software interested in the workflow may read from the Workflow Storage to observe the current workflow status or to examine a Material's History. Our Workflow Storage provides methods to

- Get a Material's State in a workflow (or all workflows).
- Get a Material's History (a list of Events) in a workflow (or all workflows).
- Get all Materials in a workflow that are in a certain State.
- Get all Materials of a certain type that are in a certain State.
- Get all Materials that have been associated with a certain Event. or type of Event.
- Get all Materials that have been associated with a certain type of Event with a certain Result.

We have not written a front-end to the reporting system to present this data to a user.

Users of the system will undoubtably want to perform more complex queries that combine the above mentioned criteria or that involve properties of the Materials and Events. Since our workflow system does not specify the persistence mechanism for Events and Materials, we cannot rely on its implementation and therefore have no way to query fields of these persistent objects. This presents no difficulties for the runtime components of the system but precludes a generic reporting infrastructure. We anticipate that a generic reporting infrastructure will be written to support a generic object storage facility, but have not implemented such a tool.

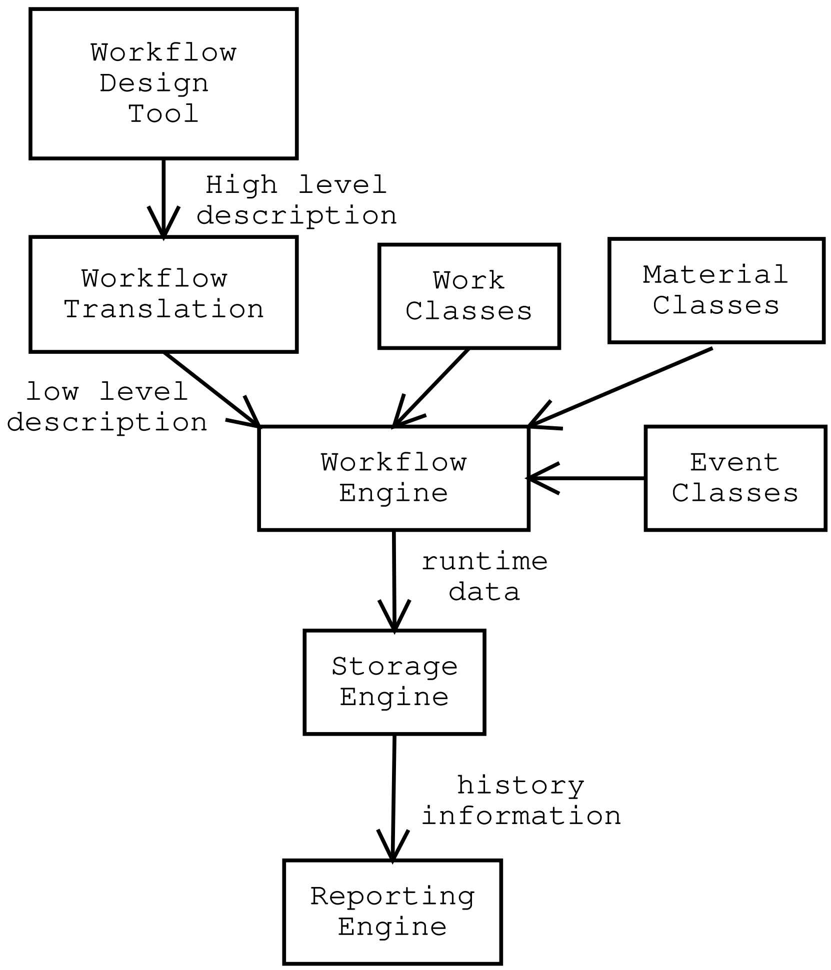

Figure 2-2: **Major System Components** The user creates a high level description of the process with the Workflow design tool (the GUI). The GUI's translation component translates the high level description into a low level description that uses only simple primitives of Listeners, Results, and States to process Materials. The user provides the low level description and code to implement the Steps (work classes), Materials, and Events to the Workflow Engine, which enacts the process.
The Workflow Engine relies on the Storage Engine to maintain the State and History data. A reporting component, which we have not implemented, might read the History data from the Storage Engine to generate reports about workflow throughput or status.

Chapter 3

System Implementation

Our Workflow System's implementation consists of components (process design tool, storage, and runtime engine) that interact through a few simple interfaces. The Workflow Description Classes lie at the center of our system. These objects describe a workflow process and all other components depend upon them. The Base Runtime Classes represent the dynamic objects (Materials, Events, Listeners) used by the other system components. The remaining components (storage component, a runtime engine, and a user interface) all interact through the WorkflowDescription classes and the Base Runtime classes. Consequently, the interfaces and implementations of the peripheral components can be changed with minimal effect on the remainder of the system.

Figure 3-1 shows the flow of data through the Workflow Engine.

3.1 Workflow Description Classes

The low level workflow description language's components are shown in figure 3-2. We ignore the high level description system here since many high level description languages could be translated into our single low level language and because only the user interface relies on the high level description.

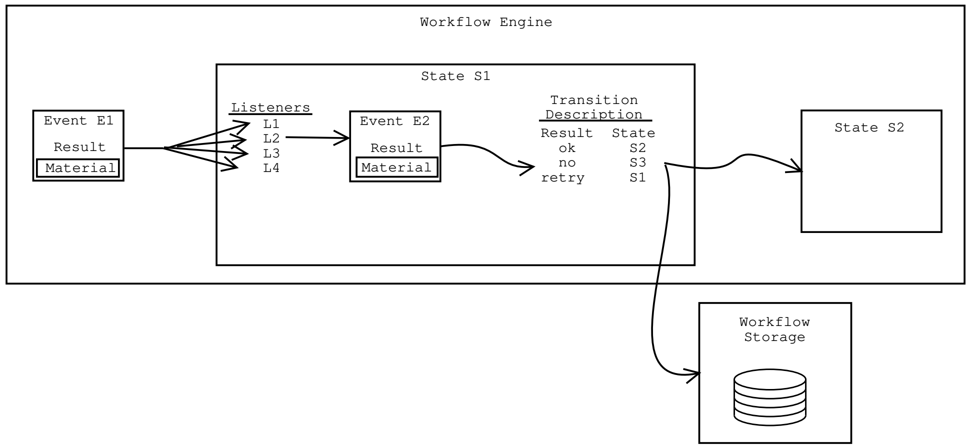

Figure 3-1: **Runtime Dataflow in the Workflow Engine** This figure illustrates the flow of data for one Step in a process. The Engine has already read a process description and created its runtime representation of the States in the process. At left is the incoming Event, E1, which contains a Material. The Engine has used the Result of E1 to route it to State S1. At S1, the system invokes each Listener on the Event (and therefore on the Material). One Listener may produce a new Event, E2. The Engine uses the Transition Description associated with S1 to map E2's Result to the next State. For example, a Result of "ok" maps to State S2. Before dispatching E2 to S2, the Engine uses the Workflow Storage to associate E2 with the Material and to associate the Material with State S2.

WorkflowDescription `WorkflowDescription` is the top level object to describe a process. A `WorkflowDescription` has a `Name` attribute and contains a collection of `State`s.

State The `State` class corresponds directly to the concept of a State as a queue of Materials waiting for some action or as a label to describe a Material's status in the process. A `State`'s `Name` attribute must be unique among all States in the WorkflowDescription. The State also has boolean `Initial` and `Terminal` to indicate whether the `State` is the first or last in the workflow. A `State` contains a reference to a single `TransitionDescription` describing the path of Materials that have been processed by the State. Finally, a `State` contains a set of `ListenerDescription`s, at most one of which may describe an `ActiveListener` (see section 3.2.4). A `ListenerDescription` specifies the `Name` of a class to execute when

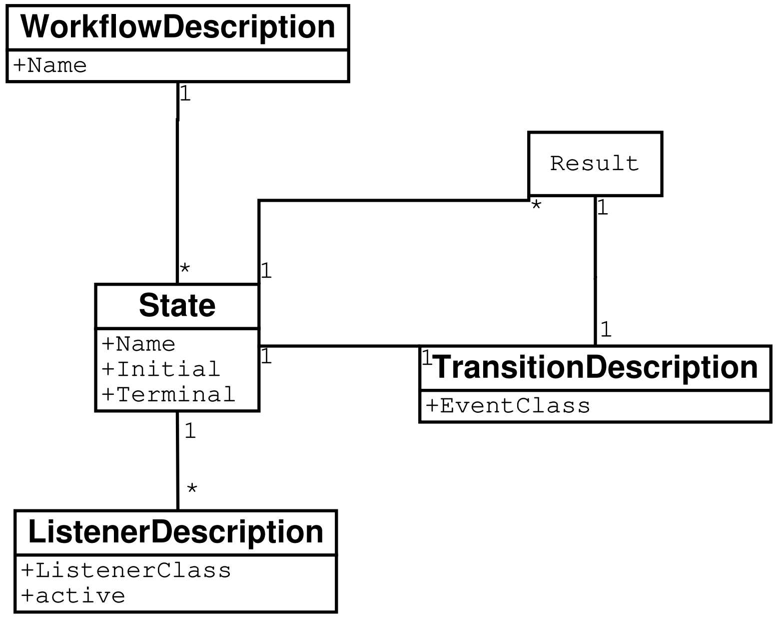

Figure 3-2: **UML Diagram of the low-level Workflow description classes**. A WorkflowDescription is a collection of State objects with a Name. The WorkflowDescription may contain any number of States, but a State must be contained by exactly one WorfklowDescription. A State has a name and boolean attributes to indicate if it is an initial or terminal State in the process. A State may contain any number of ListenerDescription, each of which describes one Listener to be run on Materials in that State. At most one of the Listeners at the State may be marked "active" to indicate that it creates a new Event when invoked.
A non-terminal State must contain one TransitionDescription. Terminal States have no TransitionDescription. The TransitionDescription indicates the class of the Event object that Listeners at the State may produce. The TransitionDescription also maps the Result of the Events produced to another State.

a Material arrives in this `State`. A `ListenerDescription` also contains a set of parameters that can modify the behavior of the `Listener` code on a per-workflow basis. A `State` does not contain the actual Listeners; this would require having working code for those classes during process design, which may happen before the Listeners have been written.

TransitionDescription A `TransitionDescription` describes the paths from one `State` to other `States`. The `TransitionDescription` specifies the `Class` of the Event that will be passed from the `SourceState` to the next `State`. All `Events` contain a

`Result`, a `Material`, and a `Date`. The `TransitionDescription` maps each Result to the next `State`.

3.2 Base Runtime classes

All runtime components share the Base Runtime classes. Materials represent the physical or computational objects that the system processes. Events record the results of the work performed by Listeners on these Materials.

3.2.1 Materials

Material classes implement the `Material` interface. A Material must have

- A unique numeric identifier. The Workflow system uses the identifier internally to track Materials. To keep the History and State for a Material, the system must have access to some globally unique identifier for that Material.

- A creation date. The creation date is primarily useful to end-users or to developers who are debugging the workflow.

Classes that implement the `Material` interface may have as many other fields as the developer sees fit to represent the physical or computational object.

3.2.2 Events

Events implement the `Event` interface to record the actions in the Workflow. Every Event has

- A unique numeric identifier. The system uses the identifier to associate the Event with the relevant Materials.

- A Date on which the action took place. In some cases, a single date is insufficient to describe an Event (e.g. the starting data and ending date of a long running computational process might differ by many days); we use a single date now for simplicity. Event classes are free to define attributes to allow more temporal information to be stored.

- The Material upon which the action was taken.

- The Result of the action. This is a simple summary (represented as a string) of the outcome or Result such as OK, PRESENT, or RETRY.

Classes that implement the `Event` interface may have as many other fields as the developer sees fit to represent the physical or computational work they represent.

3.2.3 `Storable`

Materials and Events share three common behaviors in the context of a workflow system. First, both are passed as inputs and outputs to and from the code that implements the actual work of the process. This work may be done across time and machines. Second, relationships between both types of objects are stored by the workflow system to indicate the History and state of the system. Third, both Events and Materials are assumed to have fields of interest to the user but which the generic system components can ignore.

To recognize these similarities between Events and Materials, both interfaces inherit from the `Storable` interface. Storable objects provide two behaviors. First, each Storable object has a unique numeric identifier that the workflow system can use to track the object. Second, a Storable object supports the `store` and `get` operations that save its fields to some persistent store and retrieve those fields.

To the runtime components of the workflow system, the methods used to generate unique identifiers and to provide object persistence are largely irrelevant. The system's Engine calls the `store` method at appropriate times to ensure that objects are stored However, the system does not attempt to access the stored data nor does it use the `get` method.

3.2.4 Listeners

The work classes implement the actual work of the Workflow. Since our Workflow system is written in Java, we attempt to express their role in standard Java terms. Hence, we call these Listeners attached to a State. The Events received indicate that a Material has entered the State. Implementing `Listener`, these classes provide an `onEvent` method that receives the Event generated by the previous Step/Action in the workflow. This input Event is generally the output of another Listener (the one attached to the previous State) or the output of an external program. From the Event, the Listener extracts the Material to be processed.

The decision to define `onEvent` as taking an `Event` as input has several important characteristics. First, this approach seems natural to implement; whatever system code for passes Events to Listeners should clearly not depend on the concrete type of the Event. Instead, it should handle any implementation of Event. A Listener does not know at compile time type what type of Event it will receive. While it would be trivial for code to use Java's `instanceof` operator or a reflection method to determine this information at run-time; however, our design must encourage the development of work classes that are independent (to the greatest degree possible) of the type of Event received and therefore of the preceding Listeners in the workflow. We hope that this independence will make the code written more general such that it can be easily reused in other workflows.

Our system provides a mechanism to parameterize the behavior of the work classes. For each Listener attached to a State, the process designer may provide a list of properties. At run-time, the code performing the action has access to these properties and may modify its behavior accordingly.

We have divided Listeners into two types. The first type of Listener receives Events, but has no effect on the system. This type of Listener is passive (from the system's view). The second type of Listener is active; it receives Events, processes Materials, and then returns a new Event describing that work to the system. A State can have only one Active Listener associated with it, whereas any number of Passive

Listeners may exist for a State. Multiple active Listeners would allow multiple results to be returned, each potentially routing the Material to a different State.

Passive Listeners

Passive Listeners can implement many key functions of the system even though they cannot directly return an Event. A simple example is an observer for an error State that emails interested parties to inform them that the processing of a Material has failed. Passive Listeners might also append items to the work list of a user. More complicated passive Listeners might interact with external databases or programs.

Active Listeners

Active Listeners provide a method to return an Event to the system. An Active Listener can represent any work that can be performed on demand. In general, any purely-computational Step meets this criteria.

External Programs

Not all work is performed by Listeners attached to States. The inherent restriction of a Listener is that it be able to perform its work upon demand- when an Event is passed to it. In many cases, a Listener would have to block until some laboratory or "real world" work is done. Other workflow systems have included protocols, often quite complex, to interact with asynchronous processes or entities (e.g., people).

In our system, we took a simpler approach. Rather than block, the Listeners do whatever work can be performed on demand but return no new `Event` to the system. Instead, some external program will provide the `Event` to the system when the work is complete. For example, the passive Listeners at a State might add an item to a work list to indicate laboratory work that needs to be done by a user of the system. The user would run a data entry program that dispatches the relevant Events back to the system. Section 3.6 discusses other uses of external programs in more detail.

3.3 Workflow Storage Component

The storage component provides persistent storage of process definitions, stores the runtime status of process instances, and records the History of process instances. The current implementation of the Storage component uses an SQL database as a persistent store. All database interaction goes through the Java JDBC interfaces to minimize the quantity of RDBMS-specific code. The database schema for our Oracle implementation of the storage component is found in appendix A.

The Storage Component presents a public interface to retrieve process definitions and to query the current State and History of a Workflow Instance.

3.3.1 Process Definition

To store a process definition, the system needs to serialize the objects described in figure 3-2. The tables `Workflow`, `WFState`, `StateListener`, and `TransitionDesc` store one row per corresponding objects. `Properties` is a supplementary table to store the properties for Listeners. `TransitionResult` corresponds to the mapping from a `TransitionDescription` to `States` and stores one row per result.

The `Class` table maps a class name to an internal id. This is used for process definition storage to record the Listeners attached to a State and to record the type of Event expected at a TransitionDescription.

3.3.2 Process Instance Status

The runtime tables store the status of the process instances. `StateMap` maps a [Material, Workflow] pair to a State. No Material can be mapped to multiple States in the same workflow.

`ClassMap` maps the identifier of `Storable` objects to their class. This table is updated at runtime as Materials are associated with States. The purpose of the information is to be able to recreate objects as necessary. For example, a program might wish to retrieve all of the Materials in a specified State to perform some processing. The system fulfills this request by examining the StateMap to obtain the object identifiers for all of the Materials in that State. It can then determine the class of each object and use Java's reflection methods to instantiate an object of that class, providing the identifier as the single argument to the constructor. The external program could then call the object's `get` method to retrieve the other fields as necessary.

The append-only table `HistoryMap` associates Events with Materials to record the History of the Material. All writing to this table occurs while the process is active.

The `WorkflowInstance` and `InstanceMaterialMap` tables are not used by the basic workflow implementation. Their purpose will be discussed as part of the versioning implementation (chapter 4).

3.3.3 Process History

Process History, as mentioned, is stored in the `HistoryMap` table. This table associates a Material with an Event. For efficiency, the Result, Date, and Workflow identifier are also stored. This allows the system to generate statistics about the number of Materials processed in some time frame or the number of Events containing Result.

3.3.4 RDBMS Dependencies

The dependencies on the RDBMS implementation are limited to

- The schema. Different database systems have slightly different Data Definition Languages. Consequently, we must write a schema for each database system. The correct schema must be used when the database is prepared for use with the workflow system. After this one-time event, the different schemas are not relevant.

- The database connection. Different database systems have slight differences in the code required to make a connection to the database. This problem can be handled in several ways by encapsulating the database dependency in a single class or method, thus insulating all but a small part of the system.

3.4 Runtime Engine

We based ur implementation on Sun's Java Message Service (JMS) specification, a set of interfaces to describe messaging services. JMS itself provides no implementation; instead, existing message broker products implement the JMS specification.

We decided to use JMS as the core of the Runtime Engine since it provided many of the necessary features. Figure 3-3 shows the relation between JMS, the Runtime Engine, and the user's Listeners. We prefer a third-party JMS implementation to a custom message delivery implementation because its authors have have already addressed reliability and scalability. A third-party messaging service also reduces the code size of the Runtime Engine.

The authors of JMS implementations have given substantial thought to reliability. JMS allows messages to be delivered in several modes, one of which provides guaranteed message delivery. Once the JMS implementation has accepted a message for delivery, it guarantees that the message arrives. JMS also tracks delivery attempts and marks messages that have been delivered more than once (generally indicating an error condition in which the first attempt to process the Event failed).

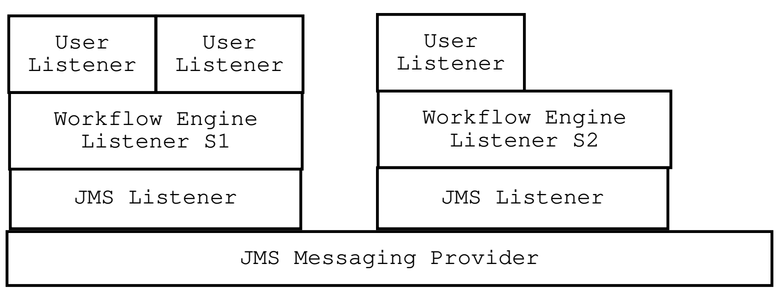

Figure 3-3: **JMS Implementation: Layers of Listeners** We implemented the Runtime Engine on top of the Java Messaging Service. JMS provides the underlying message deliver to JMS queues. Each queue corresponds to one State in the process. The JMS Listener for the queue passes the JMS message to the Engine. The Engine extracts the Event and Material from the JMS message and passes them to all of the Listeners for the State.
When a Listener at State S1 produces a new Event that will be routed for State S2, the Engine creates a new JMS message that contains the Event and dispatches that message to the JMS queue corresponding to State S2.

The second major advantage of using a third party JMS implementation is that the authors have given substantial attention to scalability. A JMS message broker (especially a commercial product) can to handle many concurrent connections and provide high throughput. In the JMS model, we can receive messages from a broker on any number of machines. If used correctly, this behavior should allow us to distribute the computational load of the workflow across many machines.

Messaging Model

JMS provides two messaging models: Point to Point and Publish-Subscribe. In the Point to Point model, messages are sent to queues. A program can register listeners for queues. The JMS software invokes listener automatically when messages are available or the program can poll the queue for messages. Using reliable messaging and transactions, a message remains in the queue until the receiving program marks the message reception as complete.

Publish-Subscribe organizes messages into Topics. A message is sent to a topic to which Listeners can subscribe. A listener receives all messages sent to the topic. A Listener can even request that the message broker store for later delivery messages sent when the Listener is not active.

The Publish-Subscribe model seems attractive for our use. We might create one topic for each State. We would then register each Listener for the State as a Listener for the topic. The key problem with this model is the failure behavior. Consider the case where several passive Listeners and one active Listener are registered for a State. If one of the Listeners fails (in our Java workflow system, this means that the Listener throws an exception back to the caller), the system must recognize this problem and stop the processing of the Material. However, JMS does not provide any convenient way to recall the messages that were sent to the other Listeners for the State (assuming that they haven't already been processed). As this violates our design goal for error handling, Publish Subscribe seems unsuited for our engine.

Another drawback of Publish Subscribe is that it may allow skew in the execution of the Listeners. Consider a linear progression of States, each with several passive

Listeners and one active Listener. If the active Listener of the first State is executed first, it sends an Event on to the next State. Depending on the JMS implementation and the State of the message broker, the active Listener in the second State may be executed before the passive Listeners of the first State. Though not addressed in our design goals or requirements for the engine, this seems to violate the expected behavior of the system.

The Point to Point messaging model meets our requirements better than Publish-Subscribe. Each State corresponds to one queue. The engine software registers special Listeners for each queue. When one of these Listeners obtains a message (either by being invoked automatically or by polling for the message), it calls all of the passive Listeners and then the active Listener, if it exists. If all of the Listeners executed successfully, the system obtains the Event from the active Listener and dispatches it to the next State. At this point, the work for this State is complete and both the message reception and message dispatch transactions may commit. If the any of the Listeners failed, then the system does not execute the other Listeners and can dispatch the Material off to the appropriate error State.

JMS Implementation Specific Details

The key functionality missing from the JMS specification is an administrative interface for creating or configuring queues and topics. Consequently, JMS implementations tend to provide methods to create queues but the methods are not standardized. As with RDBMS specific code in the storage component, we can wrap the JMS implementation specific code in one class. Each such class implements a single interface such that JMS providers all appear the same to the system.

The interface `WFEJMSAdaptor` provides three methods:

parseArgs allows the using program to pass command line parameters that the adaptor can use to determine the broker to connect to or the username or password.

getJMSConnection returns a QueueConnection to the appropriate JMS message broker. This encapsulates JMS implementation specific code for connecting to a broker.

createQueue creates a queue. This is here since to correct the lack of administrative functions in the JMS interfaces.

These methods mask the differences in JMS implementations; unlike the SQL adaptor, no `getJMSType` method is necessary to inform other classes what provider is being used.

SQL Implementation Specific Details

To mask the differences between SQL implementations, the workflow engine also defines an SQL adaptor interface, `WFESQLAdaptor`. This interface provides three methods:

parseArgs allows the using program to pass command line parameters that the adaptor can use to determine which database to connect to.

getSQLConnection returns a `java.sql.Connection` object to the appropriate database.

getSQLType returns a string describing the type of SQL database to which the connection was made.

Note that this does not cover all of the SQL implementation-specific behaviors of SQL databases that we have described. However, this is sufficient for the workflow engine; the engine itself has no direct interaction with an SQL database. Instead, the engine uses the storage component for all database interaction. The engine instantiates the storage component, passing the Connection and the string specifying the database type.

3.5 Workflow Design Tool

Our system includes a workflow design tool that provides a natural interface to the process definition and structure. We believe that the most natural representation for process structure is a modified flowchart or graph. The structure of the graph is extremely similar to the low level description except that States are replaced by widgets. Each widget may represent a simple State or a more complex entity. In figure 2-1, the AlignFail widget represents a simple State whereas the blocking widget is a high level widget that requires translation.

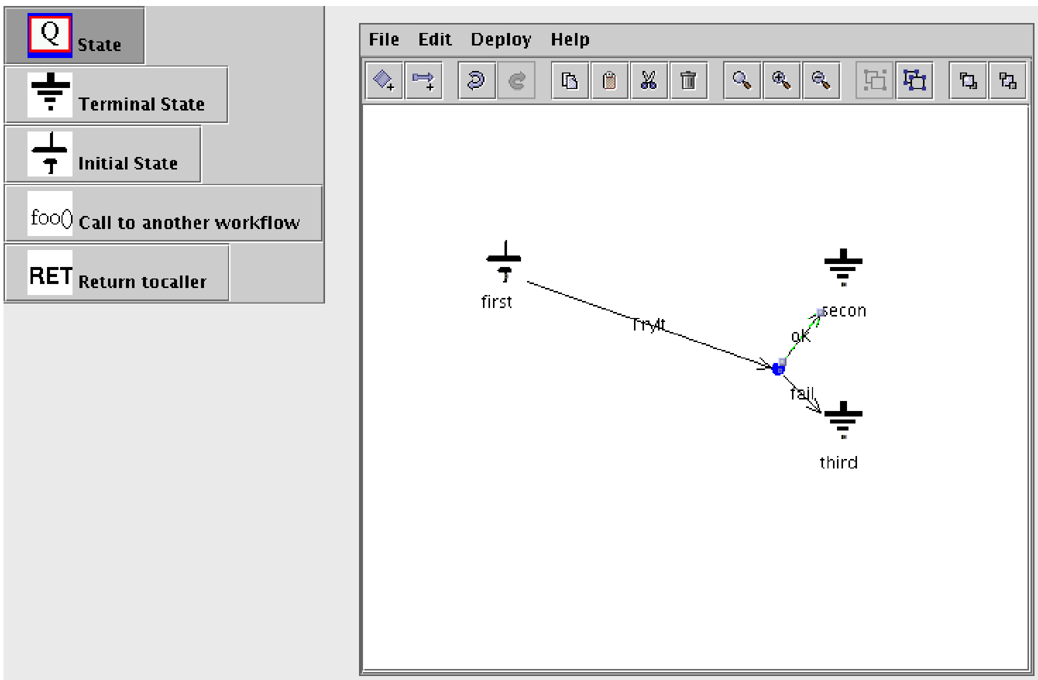

Figure 3-4: **Process Design Tool Screenshot** The process design tool uses two windows. The window shown here to the left allows the user to select a widget factory. Factories for simple States, initial States, terminal States, sub-workflow calls, and workflow returns are shown.
The window to the right contains the process design area. This process consists of one initial State, "first," and two terminal States, "second" and "third." The Listeners at "first" produce "TryIt" Events.

User-provided factory classes produce the widgets that can be placed in the graph. In figure 3-4, the window on the left shows the factories. The window on the right is the process design area. The factories may control the visual appearance of the widget by setting an icon or renderer. We based the process design area on the

JGraph[1] package.

Each factory may provide a set of filters. The filters from all factories are applied in an arbitrary order to the objects representing the high level description. The filters for a factory reduce that factory's widgets from the high level description to a low level description. When all factors have been applied, the high level process description has been transformed into a low level description. For several examples of this method, see section 3.6.

3.6 Implementing Translation

We have claimed that our core Workflow System supports complex features while implementing only a simple process description language. Successively applying the filters provided by the high level widget factories reduces high level features to their low level implementation. We now provide several examples of this process to demonstrate the implementation of sub-Workflows, Workflow chaining, blocking, and scheduling, and non-determinism in choosing the next Step.

3.6.1 Sub-Workflow

One stated requirement for the system was the ability to nest workflows to allow workflow B to appear as a Step of workflow A. In our system, this can be accomplished by writing a Listener, call it Dispatcher, which sends Events to another workflow. We provide parameters to the Listener to tell it the name of the workflow (and perhaps the name of the initial State) to which the Events are to be sent. The terminal States of the sub-Workflow contain a Listener that returns Materials to a super-workflow if one exists. To support this inter-Workflow Material passing, we make three small additions to our system: an explicit Workflow Instance interface, a means by which a Listener can obtain a Workflow Engine instance, and a means by which a Listener can obtain a Workflow Storage instance. Listeners may now imple-

[1] JGraph can be obtained from http://www.jgraph.com

ment `AcceptsWorkflowEngine` and `AcceptsWorkflowStorage` to indicate that the Workflow Engine should provide the appropriate object to the Listener. We provide a standard Listener that uses `WorkflowStorage` to examine the `WorkflowInstance` object and uses the `WorkflowEngine` to dispatch Materials and Events back to the calling workflow.

Dispatch to Sub-Workflow

To dispatch a Material from a parent Workflow to a sub-Workflow, we use a passive Listener that accepts runtime parameters to specify the target Workflow and uses the Workflow Engine to dispatch the Material. An alternate might would rely on an external program to poll the appropriate State of the parent Workflow and perform the dispatch as Materials are available. The Dispatch Listener potentially provides better performance and does not increase our reliance on external programs.

Return to Workflow

As with the dispatch, we choose to implement the return with a special Listener. However, this Listener cannot obtain the identity of the Workflow to which it must send the Material via runtime parameters as many parent Workflows could dispatch Materials to the same sub-Workflow. The Listener might examine the Material's History using a Workflow Storage object, but examining the History will be error-prone if several Workflows have processed the Material in parallel.

To correct this deficiency in our system, we add an object to represent a Workflow Instance. The `WorkflowInstance` interface (and corresponding table in our persistent storage managed by the `WorkflowStorage`) has the following attributes:

InstanceID an internal identifier for the instance in the persistent storage.

Workflow a pointer to the WorkflowDescription of which this is an instance.

Parent a WorkflowInstance of which this instance is a child.

We also add a mapping between WorkflowInstance and Material to our storage that works in the same was as the Material to State mapping. Finally, we add a

`Source` field to Event to store the identifier of the `WorkflowInstance` in which the Event took place. By setting the `Parent` field on a call to a sub-Workflow, the terminal Listener of the sub-Workflow can easily determine the return target (much as in a procedure call stack).

Our system can now maintain the information necessary to nest workflows. We must now determine how this information is to be updated and accessed. First, we add two methods to the `WorkflowStorage` interface. The first method takes a Material and a WorkflowDescription (and an optional Source that we will describe later) to create a new Workflow Instance. The second method returns the current Workflow Instance for a specified Material and WorkflowDescription. For efficiency, we provide two versions of these methods. One version returns WorkflowInstance objects whereas the other returns the identifiers for the objects.

Ideally we would like to minimize the role of user code in maintaining the `Source` field while still making it available for use. Consequently, our implementation of the runtime engine sets the source field and ensure that it is passed between Events without intervention from user code. When an initial State receives an Event, the system automatically creates a WorkflowInstance. In the remaining States, the system ensures that the WorkflowInstance in the outgoing Event is the same as that in the incoming Event. The system does not perform any checking if the Event was received from an external program, although this checking would not be hard to implement.

The creation of new WorkflowInstances at the initial State of a workflow cannot be done blindly. We use two cases. If the incoming Event has no `Source` set, then we create a new WorkflowInstance without a `Parent`. If the `Source` was set, then we change the `Source` to be a new WorkflowInstance whose `Parent` is the old source. This is the case where the Event has been sent from another workflow. By setting the `Parent`, we have stored the necessary information to send the Material and terminal Event back to the correct workflow.

3.6.2 Workflow Chaining

Workflow chaining reuses the Dispatcher Listener used for sub-Workflows. A third runtime parameter instructs the Dispatcher not to set the Parent field of the new `WorkflowInstance`. When the Material reaches a terminal State of the second Workflow, the terminal Listener cannot return the Material to the first Workflow.

3.6.3 Blocking and Scheduling

Blocking and scheduling both affect the timing of a Material's progress through a workflow. Scheduling describes any time-based influence on the progress; blocking is a specific case of scheduling in which a Material is held at a certain point until a precondition is met.

Our implementation of scheduling uses a special Listener, an external data store, and an external program (figure 3-5). The Listener adds information to the external data store about the Materials received. The external program polls the store periodically. If any of the Materials listed meet the scheduling criteria, the program dispatches them back to the Workflow system for further processing.

A simple form of scheduling allows Materials to proceed based on the time of day. For example, we might wish to run computationally intensive Steps at night. The Listener adds the Material and a desired time to the external store and the external program (perhaps run from the Unix `cron` program) polls the store and runs Materials as their specified time is reached.

3.6.4 Non-Determinism

A real-life process may involve ambiguity or parallel actions. To model these processes, we use a set of States in the Workflow. As shown in figure 3-6, we can model ambiguity in which Step should be performed by adding a pseudo-Step whose sole purpose is to choose which real Step should be performed next.

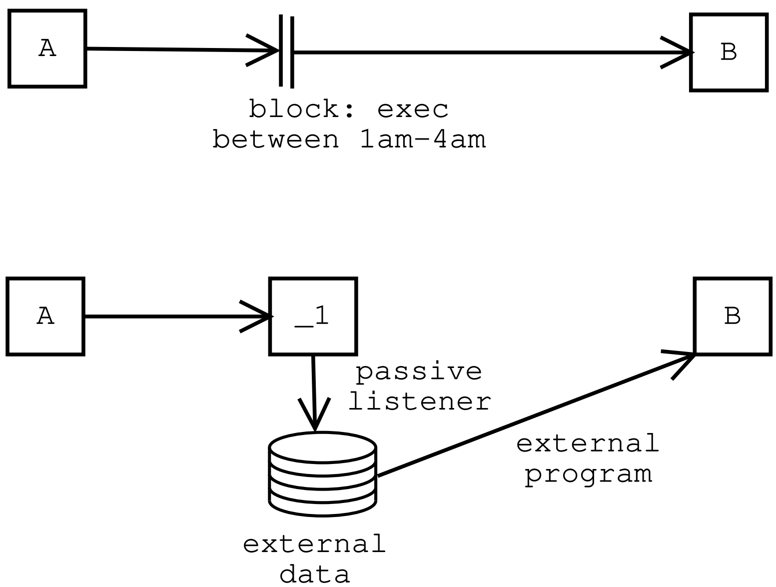

Figure 3-5: **Implementing Blocking or Scheduling** To implement blocking or scheduling, we translate the block point into a State containing one passive Listener. The Listener adds the Material to an external data store. An external program examines the store and, at the appropriate time, dispatches the Material back into the system.

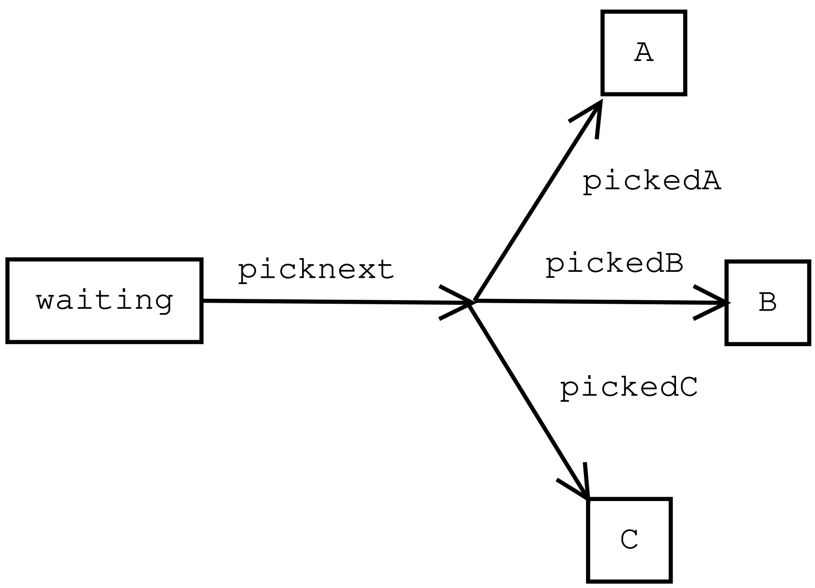

Figure 3-6: **Using Extra States to Model Nondeterminism** Special rules are sometimes necessary to route Materials through a workflow. In this example, three Steps are possible after State **waiting**, corresponding to a user choosing which to perform next. To model this situation, we have added three States, one corresponding to each possible action (A, B, or C). The Listener at **waiting** accepts input from the user to indicate which action is being performed and then routes the Material to the appropriate State.

Chapter 4

Versioning

Workflow processes inevitably change. The changes to a process definition might alter the structure of the workflow process by adding, removing, or rearranging States. The changes might modify the Listener code or the changes might alter the parameters passed to the Listeners at a State. Our system must accommodate those changes. In this chapter, we first present the types of changes in more detail and then discuss the modifications to our design necessary to accommodate change and the implementation of those modifications.

4.1 Types of Changes

Three types of changes can be made to a workflow. The first type changes the structure of the workflow process. The change may add, remove, or rearrange States or transitions in the workflow. Both the old and new process definitions can be represented using the Workflow description classes that we have previously described.

The second type of change is to the parameters passed to Listeners at run time. As with structural changes, our workflow description classes can describe both versions.

The third type of change does not modify the structure of the workflow but rather changes the code that implements the Listeners, Materials, or Events. Our framework cannot represent this type of change. We can create a new copy of the workflow description but have no way to indicate that different versions of a class should be used.

We must also consider the scope of the changes to be made. Some changes alter the process, parameters, or code for all Materials currently in the system while other changes apply only to a subset of Materials.

4.1.1 Structural Change

Workflow designers frequently make structural changes to reflect changes in a physical process. For example, researchers may replace a "dead end" that collected Materials that failed a quality check with Steps to work around the failure as soon as they develop a procedure for working around the failure (figure 4-1).

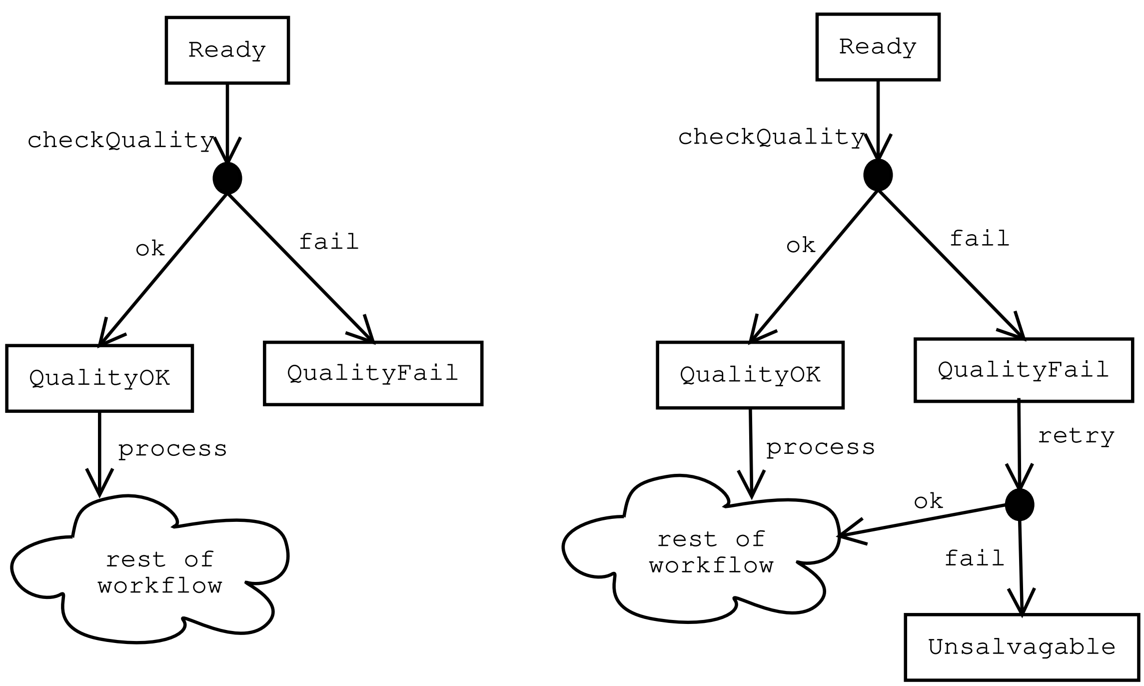

Figure 4-1: **An Example of Structural Change to a Workflow Process** Two workflow process definitions. The definition on the left is the original version. The definition on the right has been modified to include a Step to recover from the first failure of a quality check.

4.1.2 Parameter Change

The Workflow system user may change the parameters to make small adjustments to the implementation of the process without changing the code. Since the parameters that the system passes to Listeners are stored in the same way as the process structure (by the Workflow Storage Component), changes to parameters can be stored and handled in the same way as structural changes.

In the process of figure 4-1, the Listener at `Ready` that produces the `checkQuality` Event might accept parameters to specify the maximum percent of contaminant in a sample or the minimum signal intensity. Changing these parameters would not alter the structure (i.e., the image on the left would represent the structure before and after the change).

4.1.3 Code Change

The workflow designer may need to change the code for the workflow to expand the data tracked (by adding fields to a Material or Event class) or to change an algorithm. For example, the researchers using the system may wish to use an improved algorithm for examining instrument output.

4.1.4 Global or Limited Changes

Global workflow changes apply to all Materials in the system. Materials may need to be migrated from old States to new States, and all Materials should now proceed through the new process. New Materials in the workflow will also follow the new structure and use the new code. This kind of change most frequently occurs to correct a bug in the code or process to minimize the number of Materials processed incorrectly.

Global changes can be implemented by shutting the system down, running the migration tool to move Materials from old States to new States as necessary, and then bringing the system up again with the new process definition and new code.

Limited changes do not alter the process or code for all Materials. Instead, the new process and code apply to some subset of the Materials. For example, the new process might apply to Materials that enter the system after a certain date, reflecting a change to the physical process that will be made. In other cases, the change to the process applies to a subset of all Materials to reflect different categories of Materials. Consequently, a limited change means that multiple versions of the process and code may be active simultaneously. For example, two departments at a university might wish to modify their admissions procedures such that each department uses a different selection algorithm where they had previously used the same algorithm.

A limited change can be made by giving the new process definition a version number to distinguish it from old definitions and directing new Materials to the new process definition. However, using the new code is not as simple. If the workflow designer created a modified version of a Listener, Material, or Event class, the Java Virtual Machine (JVM) would not allow both versions of the class to be active at the same time because it cannot handle two classes with the same name. For example, consider a version 1 of the process that uses a Listener of class L. For version 2, we modify L. Simultaneously running both version 1 of the process and version 2 would require two different implementations (the two different versions) of L.

4.2 Design

To allow our system to accommodate structural and parameter changes, we add a `version` attribute to `WorkflowDescription`. We now require both a `name` and `version` to uniquely identify a description. A key benefit of this approach is that the `name` attribute does not change. While the system internals only use an internal numeric identifier and hence not care about the `name`, users of the system will realize that the two definitions represent variations of the same process. Furthermore, users can synchronize the version numbers with external resources such as information in a source control system.

Each version of the description can contain a different structure and different parameters. This allows the storage and runtime components of the system to operate the different versions of the same workflow at the same time by simply viewing them as separate processes.

To allow multiple versions of a class to be used in different workflows, we add version attributes to `ListenerDescription` and to `Storable` objects. The `ListenerDescription` knows the class name and version of the code that is to be used. At runtime, the system uses the name and version to locate the appropriate class. The system provides a tool to the user that accepts normal Java source code and prepares it for use with the system.

The design described thus far allows the system to store definitions for multiple versions of a workflow and to enact multiple versions of a definition at once. However, these considerations are not sufficient to perform complete History reporting. Consequently, we add `Pred` and `Succ` fields to `WorkflowInstance`. When the system ends process instance x and replaces it with process instance y (because process definition X is being replaced by Y and the Material should now proceed through Y), x's successor field is set to y and y's predecessor is set to x. This allows reporting software to associate the Events associated with a Material with the particular version of the workflow under which they occurred.

4.3 Implementation

Adding `version` attributes to the workflow description classes is trivial. Similarly modifying the storage component also requires little effort.

Accepting versioned code is more difficult. A simple approach would keep copies of different versions of the Java `.class` files as necessary. However, Java cannot load multiple versions of the same class, so there is no simple way to use multiple `.class` files for the same class. Our solution to this problem rewrites the Java source files, producing new source files that the user can compile and use without any further intervention. We will describe the details of our approach and then discuss alternative solutions.

Our system runs a source to source translation on the user's Listener, Material, and Event code and produces a mapping from the input class names to the output class names and versions. The output names and versions uniquely identify the input revision of the source code. Our workflow deploy tool automatically inserts the new class names and versions into the workflow description before writing it to the Workflow Storage. The Workflow Engine and Workflow Storage require only slight modifications to use the rewritten classes.

4.3.1 Rewriting

The rewriting process begins by identifying all files that need to be rewritten. Our implementation requires that all such files have a `.vers` extension to distinguish them from the `.java` files that are produced. This is not a requirement of the system; rather, it is a convenience to the user. The compiler sees only the `.java` files produced and ignores the input source files. Once the input set of `.vers` files has been identified, the files are parsed using the JParse[1] package to identify all classes defined in the files.

Having identified all of the files and classes to be rewritten, the rewriter must generate a unique name for each output file. It first uses an adaptor for the revision control system to discover the current version of each file. If the files do not reside in some revision control system, the rewriter assumes a version of "1.0".

Knowing the file name and version, the rewriter generates a unique name for the new file. Simply using the old name and the version does not suffice; the rewriter could be run multiple times on the same version of a file and the system cannot merely overwrite the old file with the same version. Consider a case where the code for Listener class L refers to the Material class M that it will process. Assume that both files have version 1. The system generates the unique names M_1 and L_1 and writes the output files such that L_1 refers to M_1 and L_1 refers to M_1 and deploy the resulting classes. If the developer now revises M to version 2 but does not modify L, then rewriting would produce a new class L_1 that refers to M_2. Since the system had previously deployed a class named L_1, this rewriting scheme does not work. The

[1]JParse can be obtained from http://www.ittc.ukans.edu/JParse/

rewriter generates truly unique names by generating a unique suffix for the filename as show in figure 4-2.

wflow vers.	class	version	name
1	L	2	L_2
1	M	3	M_3
2	L	2	L_2
2	M	4	M_4

wflow vers.	class	version	name
1	L	2	L_2_1
1	M	3	M_3_1
2	L	2	L_2_2
2	M	4	M_4_1

Figure 4-2: **Mapping Class Name and Version to New Name**. Both tables show a mapping from class name and version to the new class name for two versions of a process definition. The top table shows an incorrect rewriting in which both versions of the process attempt to use L_2. The bottom table shows a correct rewriting in which one process version uses L_2_1 and the other uses L_2_2.

The rewriter generates a unique suffix for the current version of each file by trying numbers starting with one until it obtains a unique file name. In the preceding example, this would have produced files named L_1_1 and L_1_2. The new file name consists of the old file name, the version, and the unique suffix, or nonce. Each name is added to a hashtable that maps old file names to the new class names and old class names to new class names. If a single source file defines multiple classes, then the same version and suffix apply to all of the classes in that file.

The system now makes a second pass over each of the files. At each point in the syntax tree for each file, the rewriter replaces the names of classes with the new names that it has generated. The rewriter then prints the modified syntax tree to a file with the appropriate new name.

The key limitation of our rewriting process is that it can only rewrite class names that it can find in the syntax tree. This includes class declarations and variable declarations. The notable case in which our rewriting fails is when a class is referred to by reflection. For example, if code were written using the `Class.forName()` method to obtain the Class object for the class of a specified name, our rewriter would not see the name of the class embedded in the String argument to `forName()`. While we might be able to create a special case in our code rewriting to handle the straightforward use of `Class.forName()`, it is not worthwhile to statically find and rewrite all code that might refer to a class name[2]. This limitation of our rewriting means that programmers cannot use reflection to refer to class names that might be rewritten. Although this limits the user code to a subset of Java, we do not feel that the limitation will be a burden on programmers.

4.3.2 Workflow Deploy

Once the source code has been rewritten, the deploy tool updates the workflow description to use the current version numbers produced by the rewriting process. The deploy tool then writes the description to the persistent process definition storage. Finally, it produce a modified version of the UI's representation that is identical to the input except that the `version` attribute of the Workflow has been incremented.

The version increment prevents the user from attempting to store the same version twice. The disadvantage of this decision is that the process definition storage may contain many workflow versions that were never used or that the user does not wish to see. This may be a particular problem during development when the programmer is constantly modifying the process definition. Although the system does not currently provide any such mechanism, it would be relatively easy to provide a tool to purge a particular version of a workflow from the system.

The advantage of automatically incrementing the version is that the programmer is less likely to overwrite a previous process definition (if the system silently allowed this to happen). Furthermore, we anticipate that in many cases, the user will want the modified process definition to be inserted as a new version. Hence, the deploy tool's behavior addresses the common use case.

[2]Consider the case where a call is made to objects for which source is not available. In theory, we could insert code into the class being rewritten to examine the Java byte code for all called objects. However, we do not consider this worthwhile for our current implementation.

4.3.3 Workflow Engine and Storage

To support versioning, the Workflow Engine must use the class name and version when instantiating Listeners. No other changes are required. The Workflow Storage must be modified to keep the version attributes. No other changes are required.

4.4 Other Approaches to Versioning

We considered two alternative solutions that do not rewrite the versioned classes but must find the correct version of the class with some other method. For example, the user or the system could maintain multiple `.class` files corresponding to the multiple versions and use the correct file at the correct time either by running multiple copies of the JVM or by loading and unloading `.class` files as necessary.

Multiple JVMs

The system could use different versions of the same class by running multiple copies of the Java Virtual Machine, where each machine would load a different version. While this would allow different versions of a class to be used at one time, the Workflow System or the user would need to store the `.class` files in different directories and then instruct the JVM to look in the appropriate places. This approach would either rely on disk storage managed by the system or require the user to put the files in the correct places. Asking the user to arrange the files violates our goal of transparency. Asking the system to manage the files adds a dependency on the filesystem.

Class file loading

The second alternative requires loading and unloading the classes as needed by the different versions of the workflow being executed. This approach requires modifying the Java class loader and synchronization between the system threads that handle Events to ensure that the correct version of the class is loaded at the correct time. This solution also requires the `.class` files to be kept somewhere in which the system can find them, creating a file system dependency.

Chapter 5

Potential Features

This chapter proposes several features for our workflow system that we have not yet attempted to implement. The first feature adds declarative constraints to the Workflow description. The second feature annotates the Listeners, Events, and Materials to ensure that all paths through a Workflow process provide the data required by the Listeners. We also propose an interface to other languages.

5.1 Constraint Checking

Constraint checking in a Workflow System prevents a transition if some value or property violates the constraint. By allowing a Workflow designer to provide constraints, the system can prevent errant data from propagating.

A frequent source of problems in workflow systems is that invalid data enters the system and cannot be easily removed. All later work with the data in the system must take into account the fact that some data is invalid. For example, a user might might enter incorrect data while transcribing information from a handwritten source or an off-by-one error might cause the wrong file to be read, misidentifying the tissue sample that is to be examined in a laboratory for signs of cancer. If the error propagates, incorrect information might be sent to a patient awaiting the lab results. Constraint checking might prevent this type of error if the intended Material meets some criteria that others are less likely to meet, for example having a “cancer suspected” attribute set.

The most likely source of the constraints is declarative statements from the developers. It may also be worth investigating automatic constraint detection that might detect anomalous results [6, 3].

Constraint checking has two uses. First, it will help developers identify bugs. Invalid data enters a system because a developer has made a mistake: data might be misformatted, values might miscalculated, or the wrong data file might be read. Constraint checking catches these errors before they propagate. This sort of checking should detect problems primarily during testing, but might catch obscure cases during actual use.

The second use of constraint checking is to detect user errors. The constraints written by a developer should accurately reflect his assumptions about the system. If a user errs and provides incorrect data, it should be flagged and rejected. In addition, constraint checking might prevent "creative" uses of a system in which a user attempts to enter some new form of data or results. While this may work in the short run, it often leads to problems when the system contains data that violates the developers assumptions. It might be interesting to apply current constraint checking and invariant detection techniques to user-supplied data.

A key component of the constraint handling will be the manner in which constraint violations are handled. The system will probably need to have different levels of constraints. Some will cause fatal errors, preventing an action from taking place. Others should provide a warning so that a user might have a chance to double-check an entry before it is committed. Rejection of user actions must be done with care so as not to alienate or offend users.

The Workflow System might implement constraint checking as another high level feature that it translates into the existing low level features of the Workflow Engine. The translation might add a State for each set of constraints and route Materials that do not meet the constrains into a special error State. Figure 5-1 illustrates the check that a tissue sample be marked as "cancer suspected" before being examined for signs of cancer.

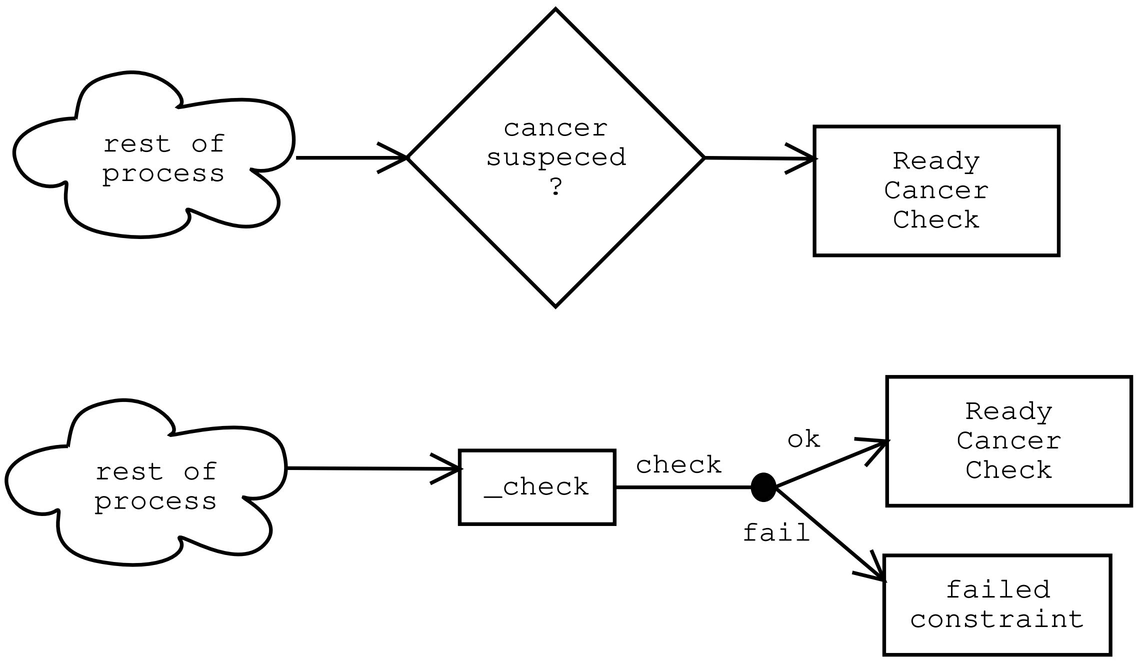

Figure 5-1: **Constraint Checking as a Translated Feature** Before undergoing a laboratory process to look for signs of cancer, a Material must have the "cancer suspected" attribute set. The process developer has implemented this as a constraint check. The diamond indicates a constraint check in the high level description. The translation adds two States. The first State contains a Listener that checks the constraint. If the Material is valid, the Listener assigns the "ok" Result and the Material proceeds through the Workflow process. If the constraint check fails, the Listener assigns the "fail" Result and the Material will go to the second added State, "failed constraint."

5.2 Data Flow Analysis

A common problem in information management systems is that step ten in a process depends on data produced by step two, but some instances of step two fail to initialize certain fields of the output Material. By the time step ten fails or produces incorrect results, obtaining the missing data may be very difficult. Consequently, we believe that a static dataflow analysis could identify paths through a workflow process that fail to meet the criteria provided for each step along the path.

As with our constraint checking scheme, we envision a system of annotations in which the code for each Listener can declare the Material and Event fields that it requires and the fields that it initializes or modifies. A static checker (used either when the Workflow is deployed or upon startup) can verify that all paths through the workflow meet the requirements for each Listener along the path.

We might extend this system by using a static analysis of the Listener code to generate the "requires" and "provides" statements. We might also use a dynamic check of the output of each Listener to ensure that it conforms to its specification.

5.3 Interfaces to Other Languages

Our current implementation of the workflow system only provides a Java API. Providing an API, or at least a gateway, to another language should be relatively straightforward.

The first step is to define an object transfer protocol between the two languages. Several forms of Object to XML serialization exist and should allow for easy transfer of the object data.

Rather than providing a complete API to the other language, we envision a gateway that accepts XML messages. The message will contain the serialized Event (and Material) and information necessary to route the message. The gateway will deserialize the XML message into the Java objects and then dispatch them using the `WorkflowEngine` interface.

Chapter 6

Conclusion

A user of our system first invokes the process definition tool, a graphical user interface, to create a flowchart-like description of a process. This high level description may include complex features packaged and distributed by anyone familiar with the system. Having described the process structure, the user writes or borrows (from other Workflows) the code to implement the Listeners, Events, and Materials. Finally, the users runs the deploy tool to rewrite the source code, translate the high level process description to a low level description, and store the low level process definition in the system's persistent storage. The user can now start the Workflow Engine and submit Materials to the process.

6.1 Evaluation

We judge our system by whether the final product meets the design requirements and by whether users agree with our assessment. We claim that our system meets our design goals; however, we do not have user evaluations to corroborate this claim.

Our first major design goal was to implement complicated, high level functions with a minimal runtime engine and translation. By demonstrating implementations of blocking, sub-workflows, chaining, and non-determinism, we showed that translation works. We anticipate that users can implement other high level features without modifications to the system.

Our core Workflow Engine implements few functions and consequently remains small compared to other systems. Its small size means that users must write more code. In particular, the high level widget factories must implement more complicated filters to translate high level features into the low level description. In practice, this does not pose a problem since the filters must be written only once and the factory can then be shared by many users. Furthermore, users may also reuse the Listeners that they have written.

Our second major goal was to handle changes to user code in a nearly transparent manner. Our process for deploying Workflow processes with versioned code meets this goal; the user provides standard Java source code and a process definition. The system handles versioning with no other interaction.

Thus far, the only use of our Workflow Management System's tools (process design tool, deploy tool, runtime engine, and storage) has been to create and run test processes. One such test process is shown in Appendix B, which describes the results of our performance testing on the system. We have also created a variety of small processes to test our high level features. Our experience shows that the tools work as anticipated, providing and easy means to define, deploy, and modify processes. However, we look forward to feedback from other users of the system.

6.2 Contributions

We have presented the design and implementation of a Workflow Management System that implements high level features using a simple core and that accommodates changes to process definitions and user code. This work claimed that high level features could be implemented with the minimal set of primitives provided by the Workflow Engine. The implementations of common features such as sub-workflows, workflow chaining, blocking, scheduling, and non-determinism support this claim. Our system also provides a transparent support for multiple, simultaneous versions of a workflow process.

Appendix A

Workflow Storage Schema

Below we include the schema used with the Oracle implementation of the storage engine presented in section 3.3. The tables presented here correspond closely to the workflow description objects and to the process State and History information.

```
-- sequence for generating unique IDs
create sequence WorkflowSequence increment by 1;

-- represents a version of a class
create table Class (
       internalID integer primary key,
       ClassName varchar(200) not null,
       Version varchar(20) not null,
       unique (ClassName, Version)
);

create table ClassMap (
       ObjectID integer primary key,
       Class references Class,
       unique(ObjectID, Class)
);

-- A schema for storing workflows in a relational database
-- corresponds exactly/very closely to the Workflow, State, EventDesc,
-- and EventResult classes in org.biojava.lims

create table Workflow (
```

```
        internalID integer primary key,
        Name    varchar(40) not null,
        Version varchar(20) not null,
        unique(Name, Version)
);

create table WFState (
        internalID integer primary key,
        Name varchar(40) not null,
        Workflow integer references Workflow,
        IsInitial integer,
        IsTerminal integer,
        unique(Name, Workflow));

create table StateListener (
        ListenerID integer primary key,
        State integer references WFState,
        ListenerClass integer references Class,
        Active integer
);

-- actual properties
create table Properties (
        Listener references StateListener,
        PropName varchar(100) not null,
        PropVal varchar(200));

create table TransitionDesc (
        internalID integer primary key,
        SourceState unique references WFState,
        EventClass integer references Class
);

create table TransitionResult (
        TransitionDesc integer references TransitionDesc,
        Result varchar(40) not null,
        State integer not null references WFState,
        unique(TransitionDesc, Result)
);

create table WorkflowInstance (
        InstanceID       integer primary key,
        WorkflowID       integer not null references Workflow,
        Parent       integer references WorkflowInstance,
        Succ       integer references WorkflowInstance,
```

```
       Pred       integer references WorkflowInstance);

create table InstanceMaterialMap (
       InstanceID  integer not null references WorkflowInstance,
       WorkflowID  integer not null references Workflow,
       MaterialID  integer not null);

-- Now schema for the runtime data structures
-- We need a table for state<->material mapping
-- and a table for history<->material mapping

create table StateMap (
       MaterialID     integer not null,
-- don't need this since a State is in one workflow and workflow + Mat
-- defines the Instnace
--       InstanceID     integer not null references WorkflowInstance,
       StateID       integer references WFState
);

create table HistoryMap (
-- material and event should probably both reference ClassMap
       MaterialID integer,
       EventID integer,
       TransitionID integer not null references TransitionDesc,
-- don't need instance since it is defined by the MaterialID and WorkflowID
--       InstanceID integer not null references WorkflowInstance,
       Result varchar(100),
       EventDate date,
       WorkflowID     integer references Workflow);
```

Appendix B

System Performance

One strength of our design is that much of the computational work can be distributed across several machines to increase the total system throughput. An ideal system's throughput would scale linearly with the number of machines added to perform the computation. In practice, the system throughput will be limited by the fact that the JMS server and SQL server must process each message. Since our system is not a commercial product, its performance characteristics are not critical; however, we wish to show that our system's design is reasonable and can handle a non-trivial load.

We performed some simple tests to characterize the performance of our system. Our test process, shown in figure B-1 contains five States and four Steps. An external program performs one Step.

Our test setup allowed us to measure the system's performance at four tasks:

1. Adding Events/Materials to the system from an external program to the initial State A.

2. Processing Events with the passive Listener at State A.

3. Retrieving Events from A, processing them in an external program, and returning the Events to the system for dispatch.

4. Dispatching Events to B and then processing them in the remainder of the workflow.

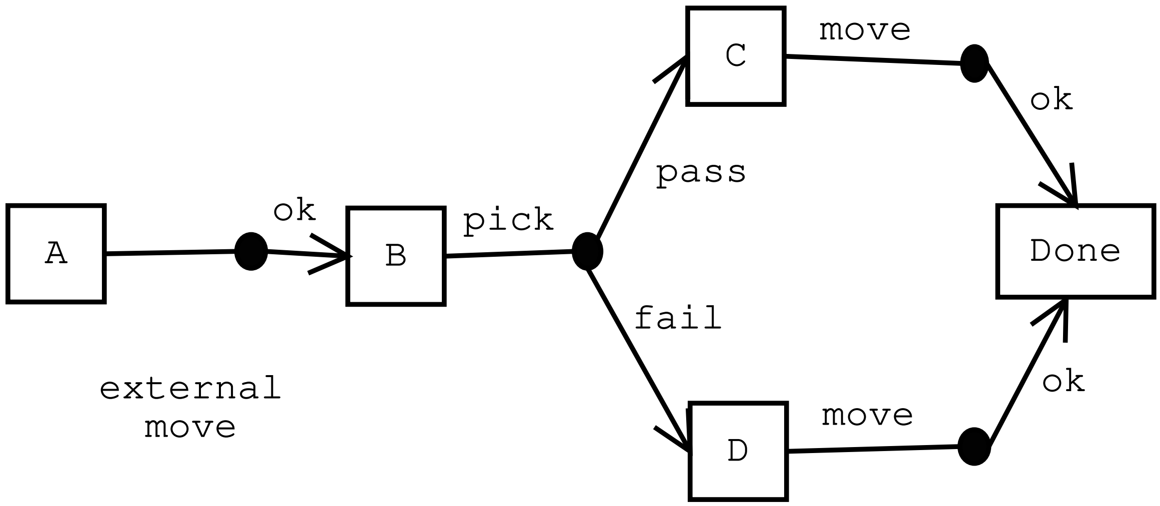

Figure B-1: **Sample Workflow for Performance Testing** The first State A has one passive Listener that prints a simple text message and no active Listeners. B has one active Listener that dispatches the Material to either C or D by choosing a Result at random. C and D each have one active Listener to pass the Material on to State Done. We wrote an external program to process Materials in A and pass them to State B.

All of our Listeners performed simple computational tasks such that the bulk of the work being performed by the system is the overhead of processing Events. We hoped to get some idea of how the system performance would scale as we processed more Events and attempted to process Events in parallel.

B.1 Dispatching Events to the Initial State

Our first test attempted to establish a baseline for the other tests. We ran one copy of the workflow engine to process Events that were in the system and one copy of the program that fed Events to State A. The program inserted 100 Events. The two relevant metrics are the time required to insert the Events and the time required to process the Events. Several repetitions of our test showed that the wall-clock time required to insert the Events was just under 90 seconds. Processing the Events also required approximately 90 seconds.

The second test ran three copies of the Event insertion program in parallel with just one workflow engine. The three insertions programs each completed in just under 90 seconds (for 100 Events each). The workflow engine took much longer to process all of the Events. This indicates that the JMS message broker was acting as a buffer for messages as expect and scaled from one to three writers with no significant performance degradation. We also see that the system seems to have high latency per message (almost one second) but high throughput. A third test using five writers and one engine confirms these observations. In all cases, the total load on the system was relatively small; as observed with the `top` program, system was not memory or CPU bound.

The fourth test ran two copies of the engine and one copy of the Event insertion program. As before, the insertion program completed in about 90 seconds. The two copies of the engine finished handling the Events in about 90 seconds as well (their completion time is clearly lower-bounded by the runtime of the Event insertion program). A test with four engines and five insertion programs confirmed this result.

The table below summarizes these results. In each row, we show the number of Event insertion programs, the number of Workflow engines, the time in seconds each insertion program took to complete on average, and the average time in seconds each Engine needed to finish processing Materials. Each insertion program inserted 100 Events.

# inserters	# engines	insert time	process time
1	1	90	90
3	1	90	283
5	1	90	520
1	2	90	90
5	4	90	130

From these results, we conclude that the JMS messaging service has a high latency and high throughput. It also buffers messages as we would expect.

B.2 External Program

The second set of tests involved an external program to process the Materials in State A, which are then passed through the remainder of the workflow. In each case, 100 items were processed. These tests also support the inversely proportional relationship between the number of engines running and the time taken to process the Events.

Bibliography

[1] Nathan Goodman. *LabFlow Manual*, 1998. unpublished.

[2] Nathan Goodman, Steve Rozen, and Alex Rolfe. *LabBase User Manual*, 1998. unpublished.

[3] Nathan Goodman, Steve Rozen, and Lincoln Stein. Workflow management software for genome-laboratory informatics, 1995. http://www-genome.wi.mit.edu/ftp/pub/papers/Y1995/workflow-grant.ps.

[4] David Hollingsworth. The workflow reference model. Technical report, Workflow Management Coalition, 1995. http://www.wfmc.org.

[5] Dragos Manolescu. An extensible workflow architecture with objects and patterns. TOOLS Eastern Europe 2001 (TOOLSEE'2001). Varna, Bulgaria, September 25-28, 2001.

[6] Steve Rozen, Lincoln Stein, and Nathan Goodman. Constructing a domain-specific DBMS using a persistent object system. In *Sixth International Workshop on Persistent Object Systems*, pages 526–541, 1994.

[7] Steve Rozen, Lincoln Stein, and Nathan Goodman. Database and workflow management challenges posed by the human genome project, 1994. unpublished.

[8] Shazia W. Sadiq. Workflows in dynamic environments - can they be managed? In *International Symposium on Cooperative Database Systems for Advanced Applications*, pages 165–176, 1999.

[9] Shazia W. Sadiq. Handling dynamic schema change in process models. In *Australasian Database Conference Jan 21-Feb 3, Canberra, Australia*, pages 120–126, 2000.

[10] Shazia Said and Maria Orlowska. Architectural considerations in systems supporting dynamic workflow modification, 1999.