1997

An Error Handling Framework for the ORBWork Workflow Enactment Service of METEOR

Davasish Worah

Amit P. Sheth
Wright State University - Main Campus, amit@sc.edu

Krzysztof J. Kochut

John A. Miller

An Error Handling Framework for the ORBWork Workflow Enactment Service of METEOR

D. Worah, A. Sheth, K. Kochut, J. Miller
Large Scale Distributed Information Systems Lab (*http://LSDIS.cs.uga.edu*)
The University of Georgia, Athens, GA 30602-7404
email: {*worah,amit,kochut,jam*}*@cs.uga.edu*

Abstract

Workflow Management Systems (WFMSs) can be used to re-engineer, streamline, automate, and track organizational processes involving humans and automated information systems. However, the state-of-the-art in workflow technology suffers from a number of limitations that prevent it from being widely used in large-scale mission critical applications. Error handling is one such issue. What makes the task of error handling challenging is the need to deal with errors that appear in various components of a complex distributed application execution environment, including various WFMS components, workflow application tasks of different types, and the heterogeneous computing infrastructure.

In this paper, we discuss a top-down approach towards dealing with errors in the context of ORBWork, a CORBA-based fully distributed workflow enactment service for the METEOR$_2$ WFMS. The paper discusses the types of errors that might occur including those involving the infrastructure of the enactment environment, system architecture of the workflow enactment service. In the context of the underlying workflow model for METEOR, we then present a three-level error model to provide a unified approach to specification, detection, and runtime recovery of errors in ORBWork. Implementation issues are also discussed. We expect the model and many of the techniques to be relevant and adaptable to other WFMS implementations.

1 Introduction

The recent push for streamlining and optimizing of organizational processes has lent to renewed interest in workflow technology. This has raised challenging requirements for WFMSs in terms of being required to support large-scale multi-system applications, involving both humans and legacy systems, in heterogeneous, autonomous and distributed (HAD) environments. Unfortunately, workflow products, in their current state, are still immature to address these emerging requirements. One of the major limitations of commercial workflow products is the lack of reliability in the presence of errors and failures [GHS95, SGJ+96, WS97, AAAM97]. In this paper, we address the issue of error handling during workflow enactment in the context of the METEOR workflow project. The related topic of workflow recovery has been discussed in [Wor97].

A *workflow* is an activity involving the coordinated execution of multiple *tasks* performed by different processing entities [KS95]. These tasks could be manual, or automated in nature. A *workflow process* is an automated organizational process involving both human (manual) and automated tasks. A *Workflow Management System* (WFMS) is a set of tools that provides support for process definition, workflow enactment, and administration and monitoring of workflow processes [Hol94]. A *workflow enactment service* consists of run-time components that provide the execution environment for the workflow process using one or more workflow engines.

The METEOR$_2$ workflow project at the Large Scale Distributed Information Systems Lab., University of Georgia (LSDIS-UGA) builds upon the earlier METEOR [KS95] effort at Bellcore. Research and development work is geared towards developing a multi-paradigm transactional WFMS capable of supporting large scale, mission critical, inter-enterprise workflow applications in HAD environments. Several workflow enactment services have been designed and implemented based on various scheduling paradigms [Wan95, MSKW96, SKM+96]. These range from highly centralized ones to fully distributed implementations using CORBA and Web technologies (either exclusively, or in combination) as infrastructure for workflow enactment. In this paper, our focus is on the error handling support provided in ORBWork, a *reliable* CORBA-based distributed enactment service for the METEOR$_2$ WFMS.

Errors are a natural occurrence in any software system; WFMSs and the workflow applications they support are no exceptions. WFMSs, in general, are complex pieces of software; the tasks (also called activities or steps) that need to be integrated by the WFMS could be arbitrary applications. When supporting enterprise-wide or inter-enterprise workflow applications, a WFMS might need to use or interact with multiple infrastructure technologies (e.g., Web, CORBA). All these factors lend to different sources of errors that need to be handled by the workflow service in a manner that conforms to the nature of the organizational process. An open-ended problem, such as error handling during workflow enactment, needs to be bounded before a viable solution can be suggested.

Error handling in WFMSs involves both specification of errors and runtime error handling policies. Most of the work discussed in workflow literature has been focused on modifying process flows to be able to deal with error conditions in a transactional manner. System level issues such as modeling and reacting to *different* types of errors (based on infrastructure, tasks, etc) has not been addressed adequately. These are issues that are crucial to real-world workflow applications.

In this paper, we present a top-down approach towards dealing with the errors in large-scale WFMS. In METEOR$_2$, we define a hierarchical set of *error classes* that is used as a basis for partitioning, detecting and handling various types of errors that occur in heterogeneous workflow enactment environments. The error classes are based on the METEOR$_2$ workflow model and are therefore reusable across all implementations of the METEOR$_2$ WFMS. We also discuss implementation support for error handling, an issue not yet discussed at comparable level of detail in the relevant literature on workflow technology. The model and design for the implementation are reusable across other WFMSs. The use of error classes as a basis for systematically modeling and handling heterogeneous errors in real-world workflow environments is one of main contributions of this paper.

In this paper, we will first describe the various types of errors that can occur during workflow enactment. This is followed by a specification of the requirements for the error handling framework in METEOR. Error handling has been a topic of research in the domains of database systems, advanced transaction models (ATMs), and transactional workflow systems. In section 4 we survey this related work. Next, we present an overview of the METEOR workflow model, the METEOR$_2$ WFMS and the ORBWork workflow enactment service. The error model for the METEOR$_2$ WFMS is presented in section 6. This is followed by a brief discussion of how this error model can be used to specify errors and error handling mechanisms at build-time. In sections 7, 8 and 9 we present a detailed discussion of task, task manager and workflow engine errors that have been implemented in the context of ORBWork. Section 10 briefly outlines the support for human assisted error handling in ORBWork. Finally, in section 11 we summarize our work and outline potential areas of research.

2 Errors during Workflow Enactment

Before we can define a scheme for dealing with errors in WFMSs it is important to understand the types of errors that can result during the workflow enactment process. In this section, we will focus our attention to workflow enactment errors; build-time errors (i.e., errors occurring during workflow design) are outside the purview of this paper. We can characterize the types of errors

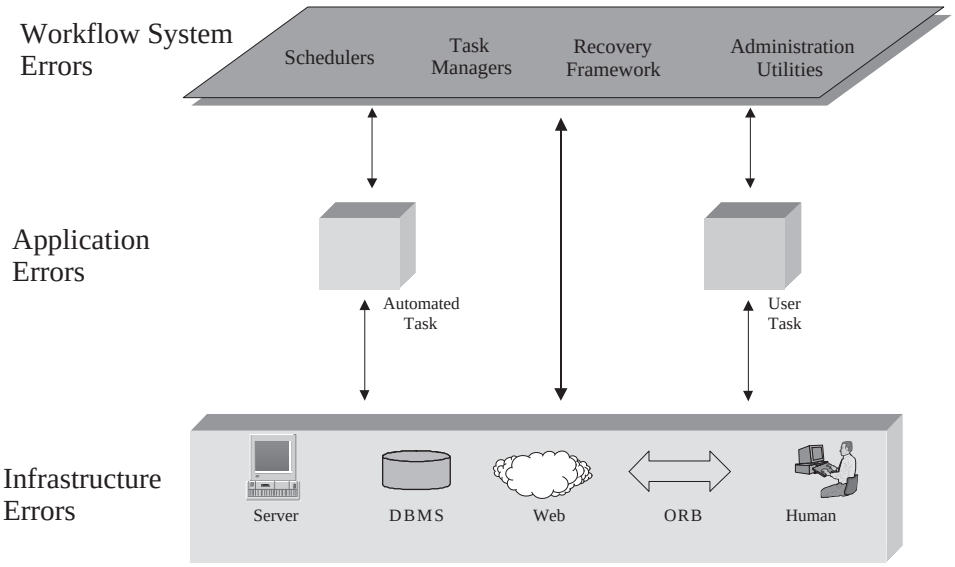


Figure 1: Workflow Enactment Errors

arising during workflow enactment into three *broad* categories (see Figure 1):

- *Infrastructure errors*: these errors result from the malfunctioning of the underlying infrastructure that supports the WFMS. These include hardware errors such as computer system crashes, errors resulting from network partitioning problems, errors resulting from interaction with the Web (e.g., HTTP errors), errors returned due to failures within the ORB environment, etc.

- *Workflow system errors*: these errors result from failures within the WFMS software. Examples of this include a crash of the workflow scheduler that could lead to errors in enforcing inter-task dependencies, errors resulting from faulty task managers, or errors in recovering failed workflow objects after a crash, etc.

- *Application and user errors*: these errors are closely tied to each of the tasks, or groups of tasks within the workflow. Due to its dependency on application level semantics, these errors are also termed as *logical* errors [KS95]. For example, one such error could involve database login errors that might be returned to a workflow task that tries to execute a transaction without having permission to do so at a particular DBMS. A runtime error within a task caused due to memory leaks would be another example of an application error.

The above categorization is a functional (descriptive) model for partitioning errors for a WFMS. Large-scale WFMSs typically span across heterogeneous operating environments; each task could be arbitrarily complex in nature. To be able to detect and handle errors in such a diverse environment,

we need a well-defined error handling approach that would allow us to specify, detect and handle the errors in a systematic fashion. In section 6 we present our approach towards dealing with workflow enactment errors.

3 Requirements for the Error Handling Framework in METEOR

The problem of dealing with errors in large-scale HAD environments poses challenging requirements. The requirements for handling errors in WFMSs are significantly different from that in transaction-based systems. WFMSs support multi-system organizational processes governed by complex business rules, and arbitrary, long-lived tasks in HAD environments [WS97]. Contrary to errors in programming languages, or structured software environments that are primarily system oriented, errors in the workflow environment involve not only simple database-like conflicts, but also organizational causes such as change in business policy, economic adjustments, role adjustments, etc. WFMSs, therefore, need a well-defined error model and an error recovery mechanism that is sensitive to the nature of the error and the context in which it occurs.

Based on our experience in modeling and development efforts for real-world workflow applications (e.g., the state-wide immunization tracking application [SKM+96]), our experience in trying to use flexible transactions in multi-system telecommunication applications [ANRS92], and our understanding of the current state of the workflow technology and its real-world or realistic applications [SJ96, SGJ+96, WS96], we formulate some of the essential requirements for the error handling framework in METEOR.

- **Support specification for error handling.** The WFMS should allow designers of the workflow process to *specify* various types of user defined-errors that might occur during enactment. For example, in the case of a task that interacts with a DBMS, it should be possible to define an error type called *login error*. In addition, it should be possible to define the error-handling policies that would be employed to deal with various task errors. For example, in the event of a *login error*, one might want to retry connecting to the DBMS with new login information for a maximum number of times. In this case, the workflow designer should support specification of task-retries with an upper limit. The error-handling framework should be flexible enough to be able to support error-handling for heterogeneous tasks and workflow enactment infrastructures.

- **Support task-specific error handling.** Error handling in WFMS should be sensitive to the errors that are *returned* by the tasks in the workflow process. The workflow engine need not be concerned with the internals of the tasks since WFMS are a means for coordination of coarse grained applications rather than fine grained programming instructions. Unlike DBMSs, the WFMS cannot *abort* a task due to *any* error that it might return. Workflow tasks, in general, are more complex than database transactions, and represent a logical activity in the overall organizational workflow. It is therefore critical to be able to detect the errors returned by arbitrary tasks and to handle them on a per-error or per-error-group basis.

- **Localize errors.** This is a feature that prevents errors in one part of the WFMS from affecting *other* parts of the enactment system. In general, it is easier to detect errors in single-process systems rather than systems that are distributed in nature. In ORBWork, we have tried to *capture* an error as close to its point of occurrence as possible; the error is then *labeled* and managed by the workflow engine in a manner that is specified by the workflow designer. In addition, we use a hierarchical error handling and failure recovery mechanism in

ORBWork. This allows us to localize errors at various stages within the workflow enactment service (see section 6 for details).

- **Support error handling by forward recovery.** Workflow applications often involve long-lived tasks. In real-world workflow applications we have seen most tasks are non-transactional, thereby not supporting the strict ACID properties of *transactions* [WS96]. Hence, although desirable, it might not be possible to recover failed non-transactional tasks using backward recovery. The use of backward recovery for most human-oriented tasks is not a viable solution since most erroneous actions once performed cannot be undone. It might be possible for the human to rectify all the inconsistencies caused due to the error and redo the actions without affecting other tasks or data objects within in the workflow; however, it would be rare to expect this behavior for most real-world human-tasks. Backward recovery is useful for purely data-oriented tasks that are transactional tasks or sub-workflows. We therefore need a forward error-handling mechanism that would semantically undo (or cleanup), or potentially undo a partially failed task.

- **Support human-assisted recovery.** WFMS are software processes. It is impossible to guarantee the success of error handling mechanism due to the undeterministic nature of errors. Therefore, the role of the human is critical for resolving erroneous conditions that could not be dealt by the error handling mechanisms of the WFMS. Also, in the case of dealing with critical errors (e.g., resulting from failure of the persistence mechanism, partitioning of the network, hardware failures, etc) the human involvement is a very important element for error handling (and also for recovery management).

4 Relevant Work

Error handling in database systems has typically been achieved by aborting transactions that result in an error [GR93]. Aborting or canceling a workflow task, would not always be appropriate or necessary in a workflow environment. Tasks could encapsulate diverse operations unlike a database transaction; the nature of the business process could be forgiving to *some* errors thereby not requiring an undo operation. Therefore, the error handling semantics of traditional transactional processing systems are too rigid for workflow systems.

Error models in most Advanced Transaction Models (ATMs) [Elm92] are restricted with respect to the ACID properties they relax. Nested transactions [Mos82] allows finer grained recovery, and provides more flexibility in terms of transaction execution. Transaction failure is often localized within such models using retries and alternative actions. We use a similar approach to localize errors in ORBWork.

A mechanism for dealing with errors in an ATM for long running activities was proposed in [DHL90, DHL91]. It supported forward error recovery such that errors occurring in non-fatal transactions could be overcome by executing *alternative* transactions. The work on flexible transactions[ELLR90, ZNBB94] discusses the role of *alternate transactions* that can be executed without sacrificing the atomicity of the overall global transaction. A workflow system that implements a flexible transaction model has been discussed in [AAA+96].

In the work on nested process management systems [CD96], the authors present a formal model of recovery that utilizes relaxed notions of isolation and atomicity within a nested transaction structure. The recovery model uses backward recovery of some of the child transactions for undoing the effects of a failed global transaction. The backward recovery approach has limited applicability in workflow environments in which it is either not possible to strictly reverse some actions, or is

not feasible (from the business perspective) to undo them since this might involve an additional overhead or conflict with a business policy (e.g., in a banking application). Sagas [GMS87] have been proposed to address many such issues in which failure atomicity requirements have been relaxed. Compensation has been applied to tasks and groups of tasks (*spheres*) to support partial backward recovery in the context of the FlowMark WFMS [Ley95]. We support compensation through the use of alternate tasks that would *undo* the effect of its failed counterpart.

Although ATMs provide models provides well defined constructs for defining alternative flow of execution in the event of errors, they are restrictive in terms of the types of activities (relaxed transactions) and the operating environment (a database) that form the long running process and therefore, do not provide the error modeling capabilities of capturing and dealing with heterogeneous workflow errors in real-world applications that METEOR is geared to support.

The role of exceptions in information systems has been discussed at length in [Saa95]. The author presents a theoretical basis, based on Petrinets, for dealing with different types of exceptions in organizational settings using a rule-based approach. In addition, a taxonomy for exceptions is presented. This taxonomy is purely driven by organizational semantics rather than being driven by a workflow process model. System-level details necessary for specifying, modeling and detecting heterogeneous workflow enactment errors is not the focus of this work.

The workflow research community, so far, has worked on addressing error handling policies [DHL90, KS95, Ley95, AAA+96, EL96] that have been based on transactional semantics. Specification of errors has been done using a *flat* error model. This approach is not scalable in the context of large-scale WFMSs that need to be able to deal with different *classes* of errors in a flexible manner. In the METEOR$_2$ approach, we provide the capability to be able to specifically categorize errors based on the context in which they occur (e.g., task, task manager, workflow engine). The error handling capability is also based on this error model and therefore can be *tuned* to react to different categories of errors. This model also provides the flexibility to specify default error-handling capability for a group of similar errors.

In [RSW97], the authors define a coordination language for specifying task composition, inter-task dependencies of long-lived, fault-tolerant distributed applications. The model supports specification of redundant input data sources, and allows tasks to terminate in one of several distinct output states. It requires usage of a task-specific cleanup mechanisms in the event of failures of non-transactional tasks. Alternate tasks can be specified as part of a compound task to alter the flow of execution of the workflow process in case of an error. However, explicit support for specification of different types of errors is missing from the coordination language.

Capturing of task- or infrastructure-errors within the workflow enactment service has been neglected by the research community. Encapsulation of heterogeneous tasks into workflow systems has been discussed in [SJHB96]; however, this work does not discuss the critical problem of reacting to task-specific errors within the task-wrappers. In the context of real-world applications, this is a critical issue that needs to be addressed. In this work, we describe an approach that we have adopted in ORBWork that enables us to *detect* such errors, and to *map* them into a form that is recognizable by the workflow enactment service. Along with the error model, we also provide implementation support for error handling in the context of a workflow enactment service, unlike most of the other related research that lacks implementation details.

5 The METEOR$_2$ Workflow Management System

A WFMS can be described in terms of its *build-time* (also referred to as *design-time*) and *enactment* (also referred to as *run-time*) components. In this section, we discuss the aspects of these

components of the METEOR$_2$ WFMS that are useful in explaining error handling discussed in the subsequent sections.

5.1 The METEOR$_2$ Workflow Model

In this section, we review the basic components of the METEOR$_2$ workflow model as discussed in [KS95] and [MSKW96]. These include interfaces, processing entities, tasks, task managers, and the workflow schedulers. Figure 2 describes the high-level modeling units of the METEOR$_2$ workflow model using Unified Modeling Language (UML) notation [Rat97].

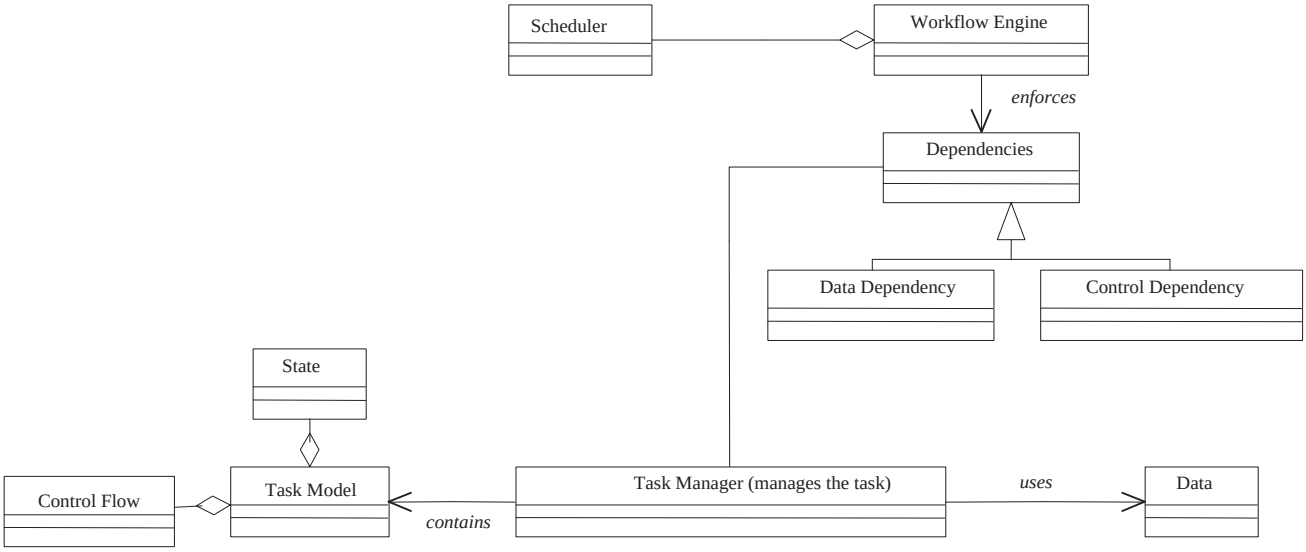


Figure 2: The METEOR$_2$ Workflow Model

Workflow. A workflow in METEOR$_2$ is defined as a collection of *tasks* and *dependencies*. It is represented as a directed graph (also referred to as a workflow map), with tasks represented by nodes and dependencies by the edges in the graph. We discuss the various types of tasks in a later section. Dependencies could be in the form of *data dependencies* or *control dependencies*. Workflows could be nested (hierarchical) in nature.

Interface and Processing Entity. A *processing entity* can be defined as any user, application system, computing device, or a combination thereof that is responsible for completing or supporting the execution of a task during a workflow execution. Examples of processing entities include word processors, DBMSs, script interpreters, image processing systems, auto-dialers, or humans that could in turn be using application software for performing their tasks.

The term *interface* denotes the access mechanism that is used by the WFMS and a workflow task initiated by the WFMS to interact with a processing entity. For example, a task that involves a database transaction could be submitted for execution using a command line interface to the DBMS server, or by using an application programming interface from within another application. In the case of a user task that requires user-input for data processing, the interface could be a Web browser displaying an HTML form, or it could be a physical switch in the case of a document management system that would activate the document generation process.

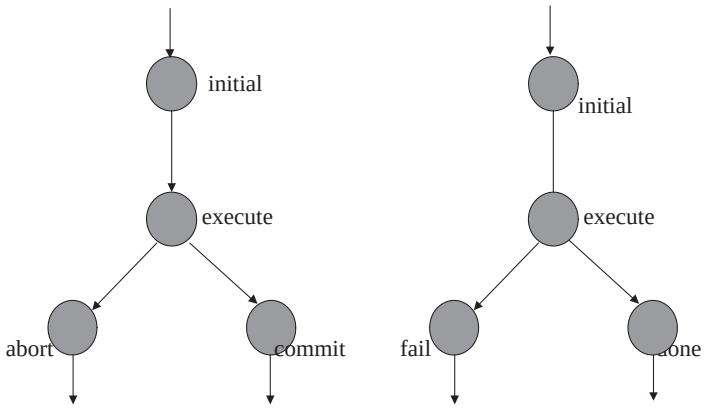
The nature of processing entities and interfaces in terms of their support for error handling and recoverability is very important when reliability of workflow execution is a concern. For example,

DBMSs support the traditional ACID transaction semantics, and hence recoverability of failed transactions is possible in transaction processing environments. However, guaranteeing failure atomicity for tasks that execute on processing entities (e.g., human tasks) that do not guarantee some or any of the ACID properties is extremely difficult.

Task. A *task* represents the basic unit of computation within an instance of the workflow enactment process. It could be either transactional or non-transactional in nature [KS95]. Each of these categories can be further divided based on whether the task is an application, or a user-oriented task. *Application tasks* are typically computer programs or scripts that could be arbitrarily complex in nature. A *user task* involves a human performing certain actions that might entail interaction with a GUI-capable terminal. The human interacts with the workflow process by providing the necessary input for activating a user task.

Tasks are modeled in the workflow system using well-defined task structures [ASSR93, RS95, KS95] that export the execution semantics of the task to the workflow level. A task structure is represented as directed graphs with the nodes denoting a set of externally visible states (e.g., initial, executing, fail, done), and the edges denoting the permissible transitions between those states. Several task structures have been developed for modeling various heterogeneous tasks [KS95, Wan95].

Transactional tasks (see Figure 3a) are used for modeling tasks that minimally obey the atomicity property and maximally support all ACID semantics of traditional transactions [GR93] (e.g., a DBMS transaction). The externally visible states of a transactional task are *initial*, *execute*, *abort* and *commit*.



(a) Transactional (b) Non-Transactional

Figure 3: METEOR$_2$ Task Structures

A *non-transactional task* (see Figure 3b) is used for modeling tasks that do not obey the atomicity property of supported by transactions. Human (user) tasks are typically non-transactional, and so are applications that interact with non-transactional resources (e.g., file systems). The externally visible states of a non-transactional task are *initial*, *execute*, *fail* and *done*.

Another special type of task structure developed in METEOR$_2$ is the *2PC* task [Wan95]. This is useful for modeling two or more distinct tasks that need to take part in a global transaction. It includes an additional task called the task coordinator that enforces the transactional context across the various tasks involved in the distributed transaction.

Task Manager. A task manager is associated with every task within the workflow execution environment[1]. The task manager acts as an intermediary between the task and the workflow scheduler. It is responsible for making the inputs to the task available in the desired format, for submitting the task for execution at the processing entity, and for collecting the outputs (if any) from the task. The task manager communicates the status of the task to the workflow scheduler. In addition, we have extended the task manager to perform error handling and recovery functions. Due to the close coupling between a task and its task manager, a task manager is termed as *transactional*, *non-transactional*, *user*, or *application task manager* depending on the task that it is managing.

In ORBWork, the scheduling mechanism (logic) is fully distributed and is embedded in each of the task managers. Task managers, therefore, perform four primary functions (for details regarding functioning of task managers, see [Das97]): (a) task activation, (b) error handling and recovery of task and its own errors, (c) logging of task inputs, outputs, and its internal state, and (d) scheduling of dependent task managers as defined by the workflow process.

Workflow Scheduler. The workflow scheduler (also referred to as the workflow engine) is responsible for coordinating the execution of various tasks within a workflow instance by enforcing inter-task dependencies defined by the underlying business process. Inter-task dependencies are specified using the METEOR designer [Zhe97]. This design is then translated into the Workflow Intermediate Language (WIL) [Lin97] which replaces the earlier Workflow Specification Language (WFSL) and Task Specification Languages (TSL) [KS95] of METEOR.

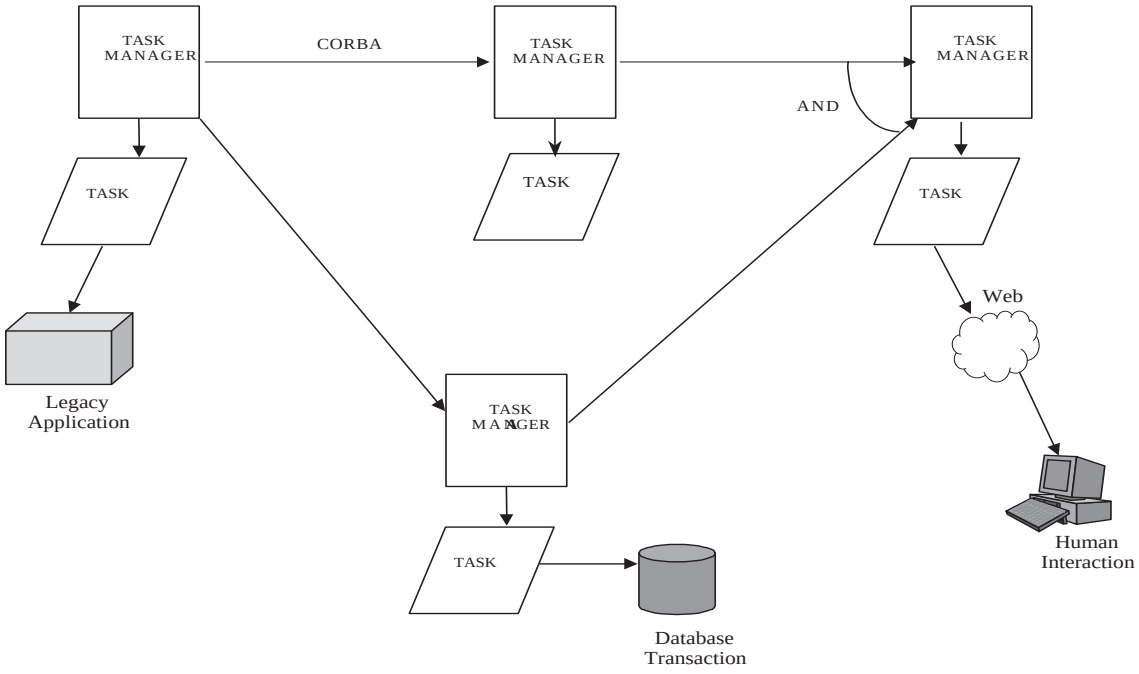


Figure 4: Distributed Workflow Scheduler in ORBWork

Various scheduling architectures have been designed and implemented [Wan95, MSKW96, Das97], ranging from highly centralized ones in which the scheduler and task managers reside within a sin-

[1] In our latest implementation, instances of the same task type of different workflow instances share a task manager to support better scalability.

gle process, to a fully distributed one in which scheduling components are distributed within each of the distributed task manager processes (see Figure 4). In this paper, we will be only discuss the fully distributed architecture that forms the basis for ORBWork.

5.2 Overview of ORBWork

ORBWork is a CORBA and Web based distributed runtime for the METEOR$_2$ WFMS [Das97]. In this section, we will provide an overview of the ORBWork enactment system. During this discussion, we will point out the contributions of this paper where appropriate.

Infrastructure. The ORBWork enactment system uses the ORB infrastructure for communication between, and distribution of, workflow components (task managers, data objects, tasks and recovery components) across host boundaries (see Figure 5). Web browsers and CGI scripts are used as a standard mechanism for user-interaction with the WFMS. Our rationale for using CORBA and Web technologies in ORBWork has been discussed at length in [SKM+96].

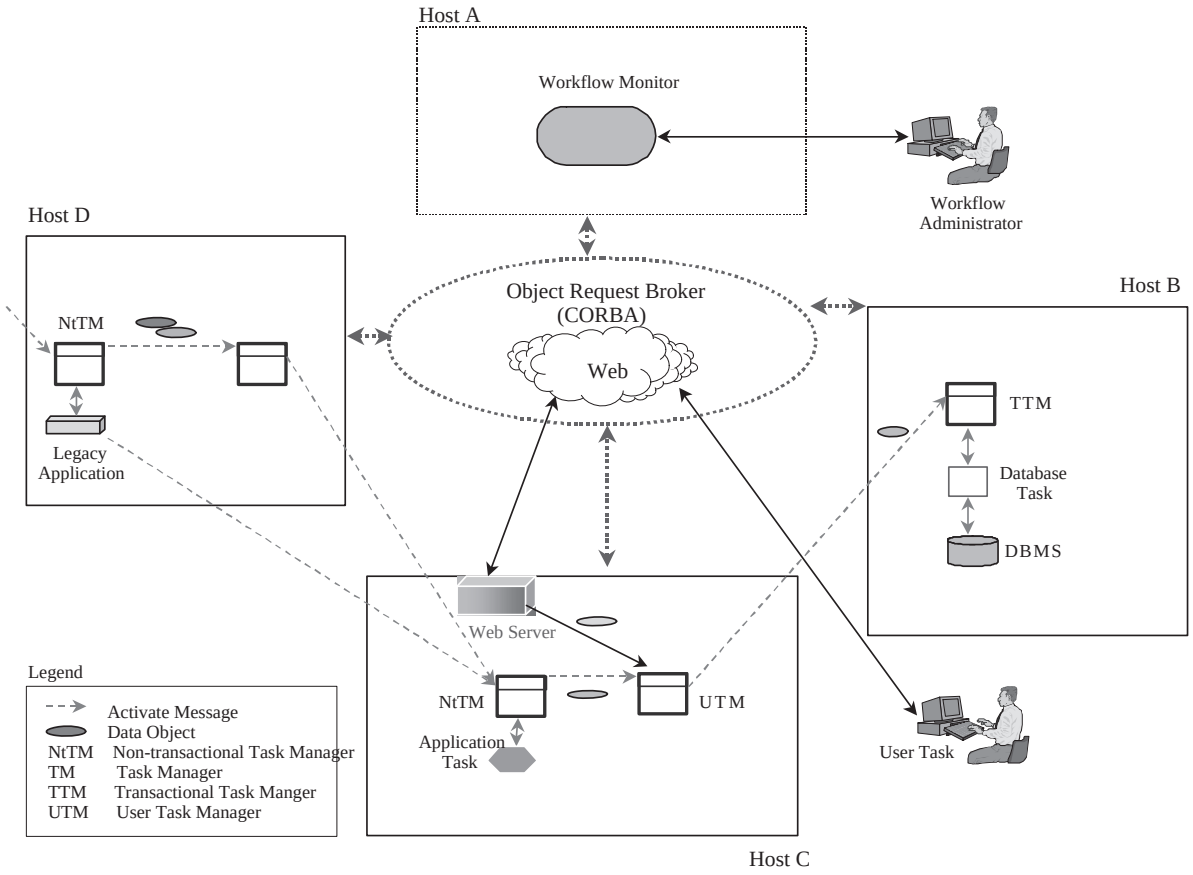


Figure 5: ORBWork Runtime Environment

Task Managers. As discussed earlier, the scheduling logic for the WFMS is distributed within each of the task managers. Therefore, each task manager performs both workflow scheduling, as well as task management functions. In addition to these, the task manager also performs error handling functionality in the form for detecting task errors, logging them, and possibly retrying tasks and performing alternate tasks. We will discuss these in detail in the Sections 7 and 8.

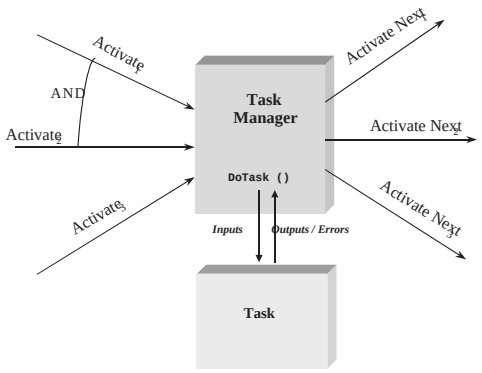


Figure 6: Abstraction of a Task Manager in ORBWork

Task managers communicate with each other via the ORB using IDL [OMG96] method invocations. Due to the location and operating system transparency offered by CORBA, they can communicate with each other from anywhere within the ORB environment by using the appropriate object names.

A skeletal version of the task manager code for a non-transactional task manager is shown in Figures 7. The task manager's code has been structured in a manner that reflects it's runtime execution model (shown alongside with the code in the figure). The error-handling has been performed using exceptions. This enables us to partition the *normal* flow of execution from the *abnormal* one.

```
// Non-transactional task manager code
/* Activate method */
void  Activate  (...) {
    try {
        try {
            // if task precondition satisfied
            if (Evaluate()) {
                        /** NORMAL EXECUTION **/
               try {
                    Execute  (...) ;// execute the task
                    Done  (...) ;   // success state
               }
                        /** ABNORMAL EXECUTION **/
               catch (METEOR2_Error meteor2Err) {
                    Fail  (meteor2Err);         // failure state
               };
            };
        catch (WorkflowError wfErr) {
            Fail  (wfErr);                      // failure state
        };
    catch (...) {
        // default error handling mechanism
    }
} // Activate
```

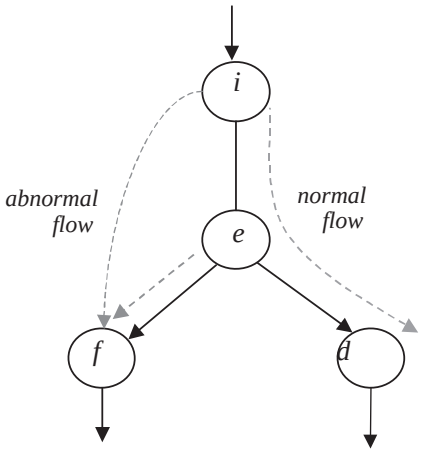


Figure 7: Basic Structure of a Task Manager in ORBWork

The design of the corresponding code for transactional task managers is similar, except for the `Done()` and `Fail()` method calls that are replaced by `Commit()` and `Abort()` respectively. In this paper, we have added an extra method called `Execute()` to enable us to keep the semantics of the code in parallel with the execution flow depicted by the task models (see figure 7). As

shown in the figure, we have separated the normal flow of execution from the abnormal flow using `try-catch` blocks (a common mechanism for exception handling in most object oriented programming languages). Any errors that are encountered in any part of the `try` block are trapped and dealt within the `catch` block. In the normal mode of execution, as shown in the figure, the task would transition through the `Execute()` and `Done()` method calls. Any abnormal (error-prone) execution would be detected through the exception handling mechanism achieved via the `catch` block.

The `Execute()` method calls `DoTask()` [Das97] to cause the workflow task to execute (see figure 11). `DoTask()` could be implemented as a wrapper for the actual task application that is executing locally on the same machine as the task manager. It contains code that represents the task application, or it might be a proxy to a CORBA *wrapper object* that is responsible for execution of an application on a remote machine. Also, `DoTask()` might *throw* exceptions denoting that an error has occurred during task execution. If this is the case, it would be caught in one of the catch blocks in the `Execute()` method.

The `Done()` method is responsible for activating the next task managers in the workflow map. It is also important to note that `Done()` does not throw any exceptions that need to be handled from within the `Activate()` method. All the errors that might arise during activation of the next task manager(s) are handled within the `Done()` call. A similar mechanism is used for dealing with errors in `Fail()`. This design makes it possible to localize the effects of the errors to logically distinct parts of the task manager code.

Types of Task Managers. Task managers that have been implemented in ORBWork include transactional, non-transactional, user, and composite task managers [Das97]. *Transactional* and *non-transactional* task managers have already been discussed in section 5.1 *User tasks* are managed by user task managers. User task managers have been specifically designed to allow humans to initiate the task activation process through the Web interface. This is achieved by using a CGI-CORBA gateway [Das97] (also known as user-CORBA gateway)[2]. User tasks have associated "to-do" *worklists* that provide a list of pending tasks for the user. User inputs form one of the implicit dependencies for a *user task manager*. User (human) tasks communicate with the task managers via HTML forms and HTTP/CGI [WWW97] functionality provided by Web servers. In our current implementation, CGI scripts are implemented as CORBA clients to user task manager objects.

Data Objects. Input and output data elements to the tasks are represented as CORBA objects internal to ORBWork. Data objects could either encapsulate data that is generated internally to the workflow process (e.g., Patient data that result from a query to a DBMS), or it could be created by the WFMS due to input from a human (e.g., data submitted via a HTML form is encapsulated in a CORBA object called a *CGI-object* [Das97]). These CORBA objects are *wrappers* around the actual data elements. This allows workflow data objects to be distributed within the ORB environment. Task managers logically enforce workflow data dependencies and pass data by exchanging the object names of the data objects.

[2]The implementation of the CGI-based user-CORBA gateway is now being replaced by Java (on the front-end) and CORBA (on the back-end).

6 Error Handling in METEOR$_2$

In METEOR$_2$, we address the following key issues to support error handling.

1. We have defined a three-layer *error model* that extends the workflow model discussed in section 5.1. This error model is used for partitioning the various types of errors that occur during runtime into three categories (see Figure 8).

2. The METEOR *task model* has been extended to a finer-granularity for modeling support so that error-specific corrective policies can be specified at build-time and corresponding error handling can be captured during enactment. We will discuss this further in the following sections.

3. To keep with the requirements for the error handling framework that were discussed in section 3, errors in METEOR$_2$ are detected and handled (when possible) as close to the point of occurrence as possible to prevent their propagation to other, unrelated components of the WFMS.

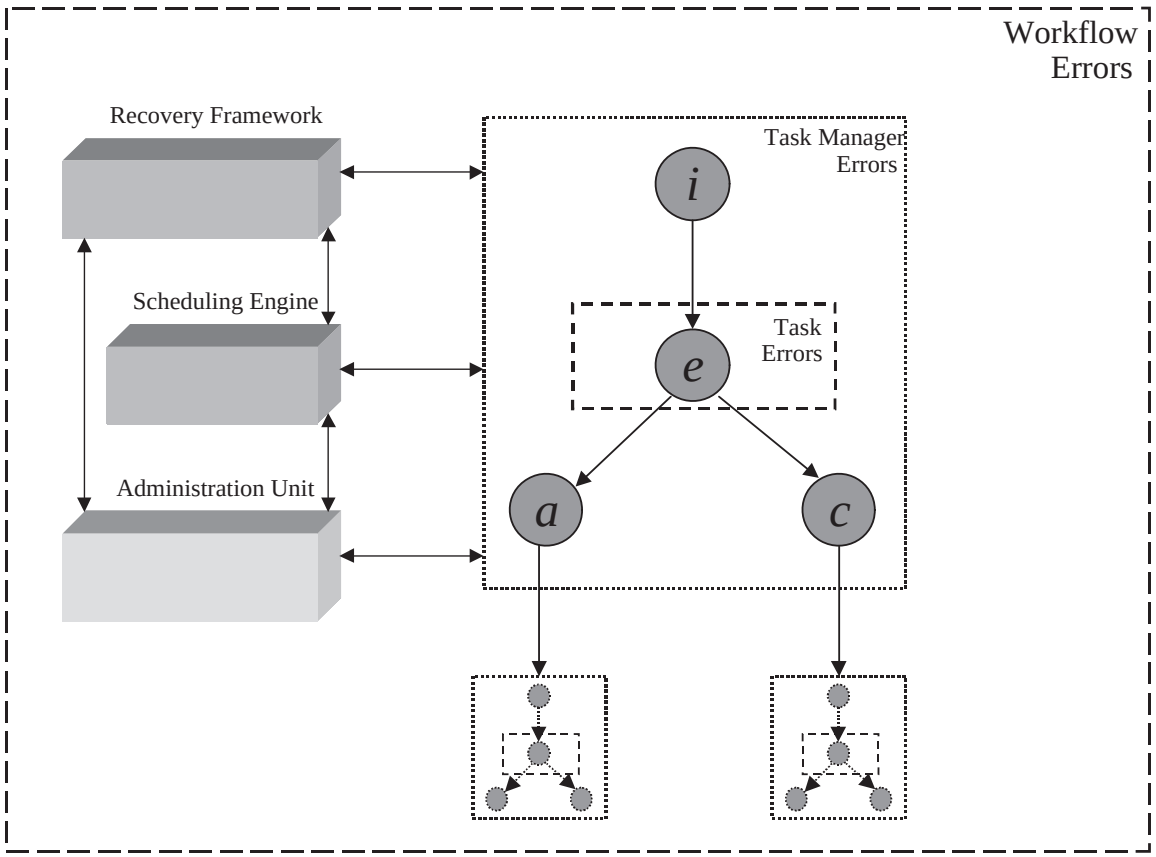


Figure 8: Partitioning Errors in the METEOR$_2$ Workflow Model

6.1 The METEOR$_2$ Error Model

In the previous section, we have characterized the types of errors that can occur within workflow systems into three broad categories (infrastructure, application and workflow system errors). To

provide adequate support for modeling and dealing with these myriad types of errors, we have adopted an error model for METEOR$_2$. We use a *top-down* approach towards modeling errors. Hence, the error model that we have adopted is based on the existing METEOR$_2$ workflow model (see Figure 2) rather than being driven by the broad category (infrastructure, application and workflow system) of errors that was based on a bottom-up approach. Infrastructure-specific and task-specific errors are mapped into our error model; this provides a unified mechanism for identifying heterogeneous errors within the workflow enactment service.

The error model is based on three enactment components of our workflow model, namely *tasks*, *task managers*, and the *workflow engine* (schedulers, recovery framework, administration units, the communication layer and other units that enable workflow process enactment). This model allows us to label errors in a manner that is independent of the workflow infrastructure and engine architecture. It also allows us to deal with errors in a hierarchical manner across the different layers (task, task manager, and workflow engine) of the workflow enactment service.

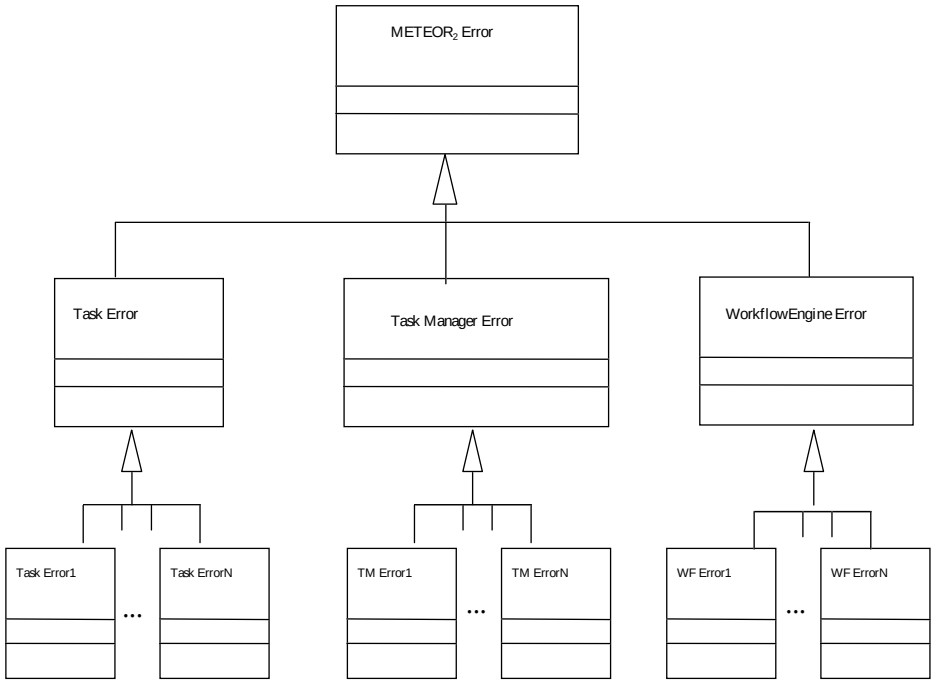


Figure 9: The METEOR$_2$ Error Class Hierarchy

The error class hierarchy (see figure 9) that define our error model is used for labeling errors at runtime. `METEOR2_Error` forms the base class from the error hierarchy. Task, task manager and workflow engine errors are derived classes for the base class object. User-defined task errors are derived from `TaskError`, all task manager errors are derived from the `TaskManagerError` class and various types of workflow engine error objects are derived from the `WorkflowEngineError` class. These classes are available with the ORBWork runtime. User-defined task errors are generated automatically from the workflow design specification (the code-generation process is beyond the purview of this paper). See section 7 for details on modeling task errors.

6.2 Implementation of Error Handling in ORBWork

In ORBWork, all errors are represented as *error classes* in the code that is generated by the workflow designer (see figure 10) The `METEOR2_Error` class forms the base class for all errors as defined in the error hierarchy 9.

```
// Base class for all METEOR2 errors
class METEOR2_Error  {
  protected:
      ErrorType   _errorType;        // enumerated type :     {TASK, TASK_MANAGER, WORKFLOW, UNKNOWN)
      char*       _description;   // description of the error
      int         _id;              // error code
  public:
    // ...
      METEOR2_Error (char* description, int id)
           : _description(description), _id(id) {}
      virtual ErrorType      errorType() { return UNKNOWN; }
      virtual char*          description() { return _description; }
      virtual int            id() { return _id; }
};

// Base class for all task errors
class TaskError: public METEOR2_Error {
      // ...
      ErrorType              errorType() { return "TASK"; }
};

// Base class for all task manager errors
class TaskManagerError: public METEOR2_Error {
      // ...
      ErrorType              errorType() { return "TASK_MANAGER"; }
};

// Base class for all workflow errors
class WorkflowError: public METEOR2_Error
      // ...
      ErrorType              errorType() { return "WORKFLOW"; }
};
```

Figure 10: Declaration for Error classes in ORBWork

In general, errors are detected closest to their point of occurrence. We have used exception handling mechanisms to deal with errors at run-time. Any piece of code (local call, remote call, or function) that is *error prone* is guarded using a `try-catch` block. A series of *catch* clauses are used to trap errors that might be *thrown* by the error-prone code. For every error that occurs, the workflow engine (task manager, scheduler, or any engine component) first tries to `catch` the specific error that might be returned.

Any error that cannot be handled by a workflow component is dealt with by a default error handling mechanism defined for all METEOR$_2$ errors. This default mechanism can be specified at build-time through the map designer. The default behavior for `TaskErrors` is to cause the task to transition to the $fail$ state; in the case of a `TaskManagerError`, or a `WorkflowEngineError` the default mechanism would be to *log* the error in the local error log and report it to the workflow administrator (see section 10.

In the next three sections, we discuss task, task manager and workflow engine errors in detail.

7 Task Errors

The METEOR$_2$ WFMS supports integration of disparate tasks (application-oriented or human-oriented) into the workflow engine. Each of these tasks might return errors, which if not handled

properly, could put the WFMS into an non-deterministic state. Hence, it is very important to *identify*, *detect*, and *handle* task errors in a manner that is based on the overall workflow process.

Task errors are *logical* errors that are reported by a task during the execution environment that is *external* to the task. Examples of task errors include exit codes returned by an application, runtime error returned by a database application on syntactic failure of a query, and so on.

Earlier, in section 2, we had discussed *application and user errors*. A *task error* as defined in the METEOR$_2$ model differs from an *application and user error* in the sense that the former is not sensitive to the internal executions of the applications, as long as they are in conformance with the workflow system, whereas the latter is sensitive to the semantics of the tasks. The problem with real-world legacy tasks is that it is not feasible to incorporate the semantics of the task into the workflow model. Therefore, we do not deal with errors that are *internal* to the task. METEOR$_2$ *task errors* denote errors (or any indications thereof, e.g., logging of errors to an application error log) that affect the external runtime environment of a task. On the other hand, a runtime error in an application leading to a *crash* would be treated as a physical *failure* by the WFMS and dealt with appropriately by the recovery framework for ORBWork (for a detailed discussion of the recovery framework, refer to [Wor97]).

7.1 Task Error Mapping

To be able to deal with arbitrary errors returned by task managers, we should be able to map them into constructs (in our case, `exceptions`) that can be manipulated programmatically and used internally by the workflow enactment service. This mapping needs to be performed in the `DoTask()` wrapper that insulates the actual task from the task manager (see Figure 11). Task-

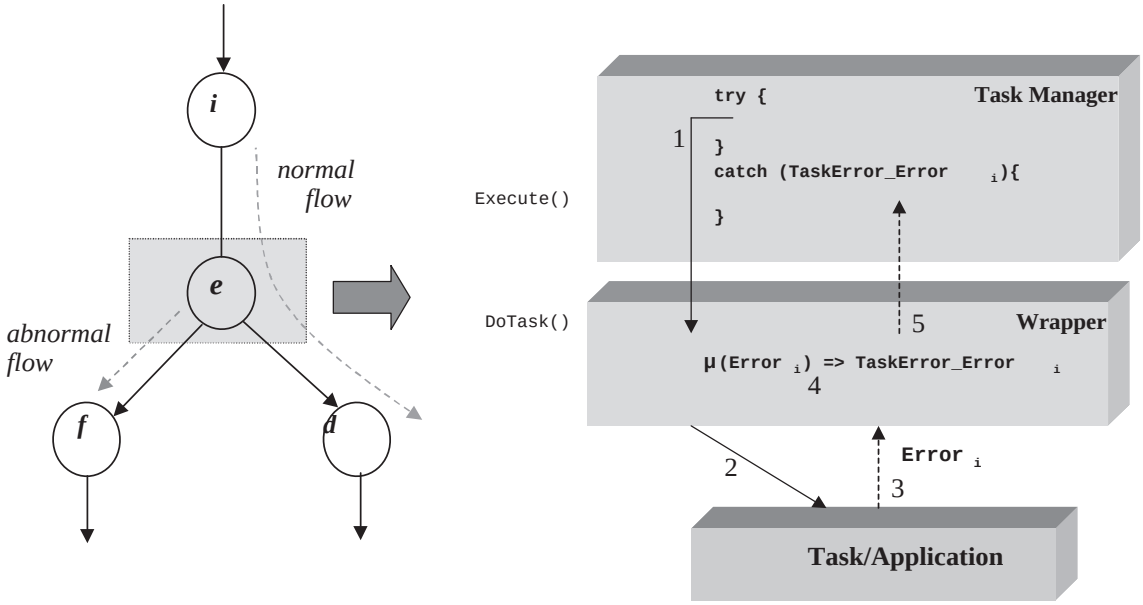


Figure 11: Task Error Detection in ORBWork

specific errors should be mapped to the specific `TaskError` defined by the user. The exact mapping (μ) to be used for performing this operation depends on the nature of errors returned by the task and needs to be performed in the `DoTask()` wrapper. Therefore, the relation μ for every task t in the workflow process can be described as:

$$\mu_t \colon T_t \rightarrow M_t$$

where $T_t = \{x \mid x$ is an error returned by task t to its task manager $\}$,
and $M_t = \{y \mid y$ is a task error defined for task t at build-time $\}$

Thus we see that the nature of μ_t is very closely related to the errors returned by t, and therefore to t itself. It is the responsibility of the implementor of `DoTask()` to define μ_t. To keep the number of elements in M_t to a tractable size, it is advisable to perform a *many-to-one* mapping from T_t to M_t.

In this paper, we have used a simple approach to define μ_t. For every error (or range of errors) specified in the task designer at build-time, it is possible to generate a unique *error id* (e.g., by using an integer counter) for internal use by the workflow engine. The *error id* is used during the runtime code-generation process to generate unique class declarations for each error type. The template for generating the user defined task errors as follows:

```
// Task error definition template.
class TaskError_<task error id> : public TaskError {
     // ...
}
```

For example, `Error_`*i* returned by a task is mapped to a *class* `TaskError_`*i* where *i* denotes the internal error id. Also note that the template ensures that `TaskError_`*i* "is a" `TaskError` due to the inheritance relationship defined by the METEOR$_2$ object-model for errors (see section 6).

7.2 Build-time Specification

Due to the sheer diversity of workflow tasks, it is impossible to predefine all of the task-specific error handling mechanisms in the workflow engine libraries. At *build-time*, it is the responsibility of the workflow designer to specify the task-specific error handling functionality needed by the workflow engine. Ideally, this detail would be specified using the task designer (and possible extensions in terms of an *error designer*). The intent of our design process is to specify the types of errors that can be returned from a task and the necessary mechanisms needed to deal with them in a manner that is consistent with the overall organizational process. Based on our earlier discussions in this paper and the need for defining flexible task-specific error handling mechanics, we have identified key information that needs to be specified at *build-time* for each task:

1. A set of errors, T_t, that are returned by a task and are of significance to the organizational process outcome.

2. A set of task errors, M_t, to be used internally by the workflow engine.

3. The mapping relation, μ_t, that would be used by the task wrapper (`DoTask()` in the case of ORBWork) to map T_t to M_t. Keeping with the graphical design paradigm adopted by METEOR$_2$, this would be done graphically; however, the description of the approach towards specification is not the focus of this paper. This relation would be stored in WIL through automatic code generation [Lin97, Das97] before it is translated to runtime code.

4. Finally, we need to specify a set of *rules* for dealing with the errors once they have been detected, transformed using μ_t, and *thrown* to the task manager's `Execute()` method or detected signal handlers in the task manager [MPS+97] (in the case of signals returned by a task).

The *rules* used for task error handling constructs in step 4 above depends on the error handling technique that is adopted by the workflow and task designer at build-time.

The approaches for defining error handling mechanisms are listed below.

Ad-hoc Approach. At the lowest level, one might want to perform error handling for errors in the *task wrapper* (`DoTask()`) itself. If dealt at this level, the remainder of the task manager code would not be informed of the occurrence of an error. It is the sole responsibility of the implementor of the wrapper to ensure correctness and reliability of the error handling procedure. At this level, the errors are in the domain of T_i; therefore, we do not anticipate supporting specific error handling mechanisms at this level.

Task Retries. The next level of error handling can be performed in the `Execute()` method. At this level, it is possible to specify a maximum number (`MAX_RETRIES`) of times a task needs to be retried if a particular error is received. For example, in the task manager code shown in figure 12, `DoTask()` would be tried a maximum of `MAX_NUMBER` times if `TaskError_ERROR1` is encountered. The boolean condition listed in the `while` clause in the code discussed above would need to depend on the number of errors for which the *retries* are being performed.

```
// Task Manager::Execute()
// called by Task Manager::Activate()
void  Execute  (...) {
        // ...
        /* set all retry counters to 0 */
        numRetries_TaskError_ERROR1 = 0;
        numRetries_TaskManagerError_TASK_EXECUTE = 0;

        // while DoTask needs to be retried.
        // Retry() uses MAX_RETRIES parameter specified in the designer
        while (Retry(...)) {
                try {
                        DoTask();                               // call the task wrapper
                        LogTaskState("EXECUTE", "NO ERROR");
                }
                /**  catch task errors    **/
                catch (TaskError_ERROR1& e1){                   // defined in designer
                        LogTaskState("EXECUTE", e1);
                        numRetries_TaskError_ERROR1++;
                        continue;                               // RETRY option specified
                }
                catch (TaskError_ERROR2& e2) {                  // defined in designer
                        LogTaskState("EXECUTE", e2);
                        throw;                                  // FAIL option specified
                }
                // ...
                catch (TaskError& e3) {                         // default
                        LogTaskState("EXECUTE", e3);
                        throw;
                }
                /**  catch task manager errors      **/
                catch (TaskManagerError_EXECUTE e4) {           // specified in designer
                        LogTaskState ("EXECUTE", e3);
                        numRetries_TaskManager_TASK_ERROR++;
                        continue;                               // RETRY option specified
                }
                // ...
                catch (TaskManagerError e5) {                   // default
                        LogTaskState ("EXECUTE", e5);
                        throw;

                /**  catch all other errors     **/
                catch (...) {                                   // default
                        LogTaskState("EXECUTE", "UNKNOWN");
                        throw;                                  // cause execution to FAIL
                }
        } //while
} //Execute
```

Figure 12: Dealing with Task Errors using Retries in ORBWork

Alternate Tasks. The third type of error handling technique could affect the overall workflow map. In some situations it might be desirable to use *alternate* tasks in the event of a failure of a particular task [KS95]. Alternate tasks could be designed to deal with specific errors. In the case of using alternate tasks, we do not guarantee atomicity or isolation of the original task under the presumption that the user realizes the implications of this substitution. In some cases

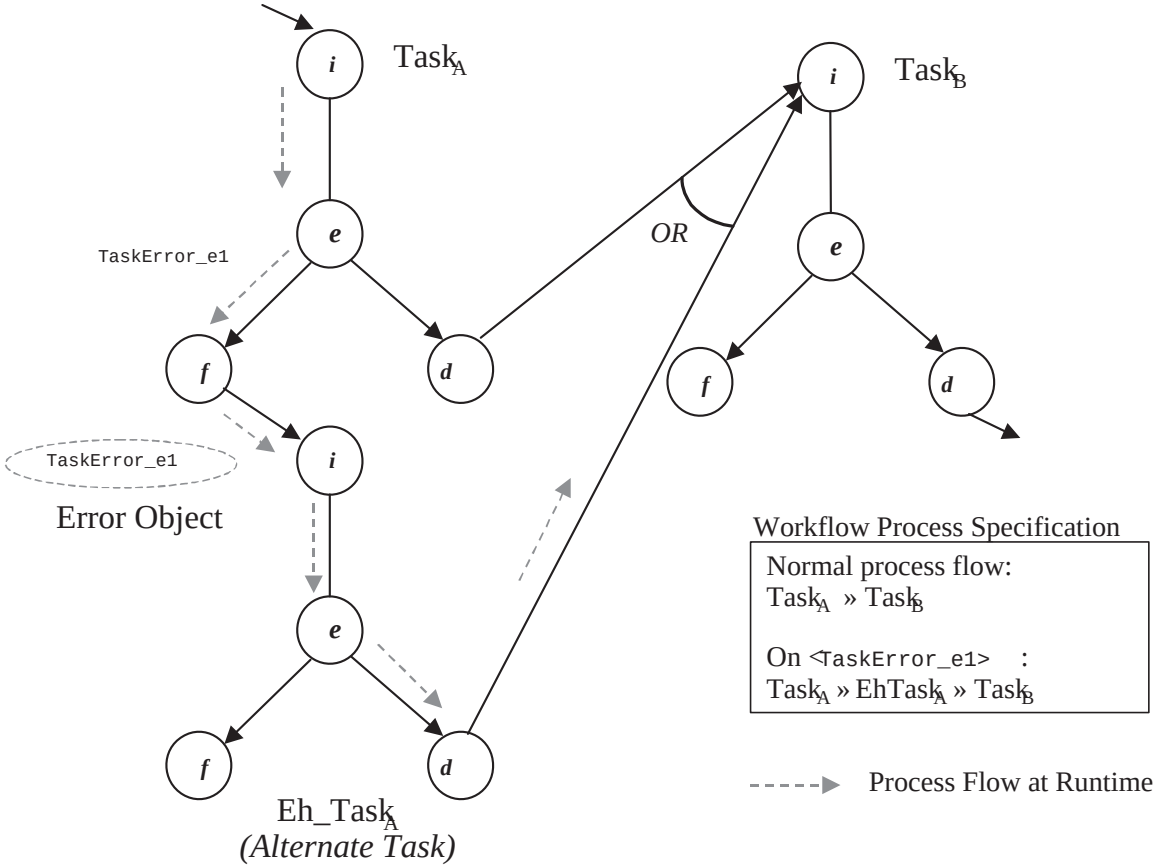


Figure 13: Error Handling using Alternate Tasks

(e.g., non-transactional task execution), it might be necessary to *undo* the effects of a failed task. This could be achieved by implementing an alternate task in an appropriate manner, and by designing the workflow map so as to activate the alternate task if the original task results in failure. Transactional tasks (e.g., DBMS transactions), due to their inherent characteristics, do not require explicit compensation. In figure 13, the non-transactional task Eh_Task_A has been specifically defined to deal with `TaskError_e1` that might arise during the execution of $Task_A$. To accommodate the effect of *repair-and-continue* for the overall workflow process, an OR relationship is established between the original task ($Task_A$) and it's error handling task (Eh_Task_A) could be defined as shown in the figure.

7.3 Task Error Handling

In ORBWork, the specification of task errors and their error-handling mechanisms maps into the final code that is generated for the `Activate()`, `Execute()`, `DoTask()`, and `Fail()` methods in the task manager.

```
// TaskManager::Fail()
// called by TaskManager::Activate() on FAILURE
void Fail (METEOR2_Error err) {

        LogTaskState("FAIL", err);                    // Log the state of the task.

        ErrorType errorType = err.errorType();        // get type of the error

        switch (errorType) {                          // ERROR HANDLING

          /* For each of the categories below perform next task activations
              specified at build-time.
             Log <"FAIL", NEXT_TASK_ID>  after each activation.
          */
                    case TASK:             // ...

                    case TASK_MANAGER:     // ...

                    case WORKFLOW:         // ...

                    case UNKONWN:          // inform administrator
        }
};
```

Figure 14: Description of a Task Manager's Fail method in ORBWork

In the event that a task error is handled at runtime within the `Execute()` method (e.g., by using retries), no exceptions would be *thrown* to the `Activate()` method in the task manager (see figure 12). In turn, the task manager would follow the *normal flow* of execution by calling the `Done()` method as shown in figure 7.

Let us now discuss the *abnormal flow* of execution in the event that a task error cannot be dealt with in the `Execute()` method. This would effectively lead to a `TaskError_<error id>` begin raised by `Execute()`. According to the code shown in figure 7, this error would be caught within the `Activate()` code block that catches any errors rooted in the `METEOR2_Error` class. In terms of the *task model*, this situation would lead to the $fail$ (**f**) state. The `Fail()` method contains the necessary constructs to deal with the particular task error by activating *alternate* error handling tasks.

8 Task Manager Errors

Task manager errors represent all errors that affect the normal mode of execution of a task manager in the METEOR$_2$ workflow model. Semantic errors within a task manager (e.g., errors resulting from faulty coding of the `DoTask()` wrapper) are outside the focus of this paper. The normal mode of execution of a task manager is discussed in section 5.1. Typical errors that are regarded as *task manager errors* include: 1) errors in preparation of task inputs and outputs, 2) errors encountered during task activation, 3) logging errors arising from recording state of a task manager,, 4) errors encountered during recovery of failed tasks and the task manager itself, 5) worklist errors, and 6) other internal errors within the task manager that might result in exceptional conditions.

8.1 Task Manager Error Mapping

In ORBWork, all task manager error classes are are declared as specializations of the `TaskManagerError` class (as shown in figure 10). We use a template-based mechanism, similar to that used for task errors, to declare task manager errors in the code for the workflow engine:

```
// Task manager error definition template.
```

```
class TaskManagerError_<tm error id> : public TaskManagerError {
     // ...
}
```

The `<tm error id>` that we have used for ORBWork is an arbitrary choice that is based on a string-to-integer mapping that we have manually implemented in this version of ORBWork. A literal string is used as a descriptive means for declaring the TaskManagerError. For example, the declaration for `TaskManagerError_TASK_ERROR` is shown below:

```
//define string-to-integer mapping
#define TASK_ERROR 1

// declare a specialization of TaskManagerError
class TaskManagerError_TASK_ERROR : public TaskManagerError {
     // ...
}
```

Types of Task Manager Errors

In our work, we have identified two basic category of task manager errors. The first category includes those task manager errors that are *independent* of the workflow implementation, i.e., they are solely based on the METEOR$_2$ workflow model. The other category consists of task manager errors that are *dependent* on the particular implementation of the depends on the architecture, infrastructure, and recovery mechanisms used.

8.2 Implementation Independent Task Manager Errors

Three of these errors are discussed in detail in this section.

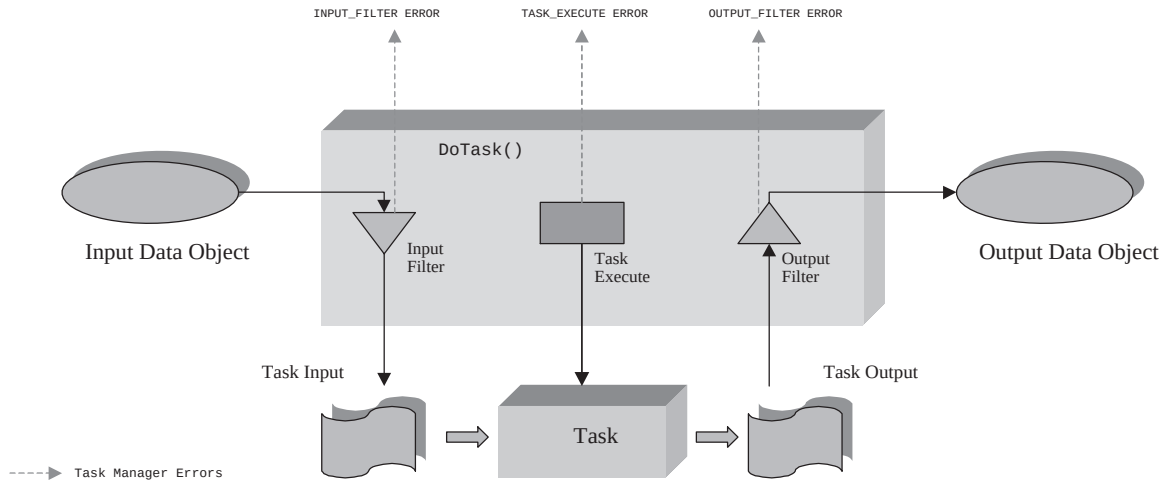


Figure 15: Task Manager Errors Raised by the Task Wrapper in ORBWork

The task-wrapper component of the task manager (`DoTask()` in ORBWork) (see Figure 15) performs low-level task manager functionality in the form of unpacking task input elements from the input-data objects and filtering them into a form that is expected by the underlying tasks, executing the tasks, and filtering the task outputs and packing them into the output-data objects in the WFMS. Any errors that are encountered while performing these functions are also modeled

as a specialization of the `TaskManagerError` object. Below we list the three types of errors that we have introduced into our model to capture these anomalies:

- `TaskManagerError_INPUT_FILTER`: This type of error represents any errors that are encountered while trying to unpack and filter the input to a task from the workflow data objects
- `TaskManangerError_TASK_EXECUTE`: Errors encountered during actual activation (execution) of a task in its native environment *and* that cannot be handled within `DoTask()` are labeled as `TaskManangerError_TASK_EXECUTE` errors.
- `TaskManager_OUTPUT_FILTER`: This exception is raised if an error is encountered while trying to map the outputs created from a task to the output *data objects*.

8.3 Implementation Dependent Task Manager Errors

The *architecture* used for a workflow engine implementation could affect the functionality of the task manager in terms of its mode of interaction with its tasks and other workflow system components [MSKW96]. For example, a fully distributed architecture would have to deal with errors resulting from network problems that would otherwise not affect centralized implementations.

Also, the workflow engine *infrastructure* could affect the types of errors that would need to be handled within task managers. For example, in the case of using a CORBA environment, the task manager would need to deal with *CORBA System Exceptions* resulting from remote object invocations [OMG96]; in the case of task managers that are implemented as CGI scripts in a Web-based workflow infrastructure [MPS+97], one would, for example, need to consider HTTP related errors that could affect the interaction between the task manager and the Web server.

In the case that extra *functionality* has been included in the task manager to support features such as recovery and monitoring capabilities, errors could arise at different stages of depending on the degree of external interaction required to support these features. For example, the recovery features for ORBWork require persistence of the task manager state at various points within the execution process. The logging procedure could result in errors that are returned from the local database that is being used. In another implementation, one might want to log states by communicating it via a transactional message queue to a remote DBMS; in this case, the task manager would need to react to transactional RPC errors that are returned by the message queue manager.

```
// In TaskManager
void RegisterWithLocalRecoveryManager (...) {
    ...
                    // try to bind to the Local Recovery Manager CORBA object
    try {
            lrmRef = LocalRecoveryManager::_bind (...);
            …
                    // register with the Local Recover Manager CORBA object
            lrmRef->Register(...);
    }
                    // catch all CORBA Exceptions
    catch (const CORBA::Exception&      excep ) {
            ...
                    // throw a semantic exception denoting a REGISTRATION failure
            throw new TaskManagerError_REGISTER (...);
    }
    ...
};
```

Figure 16: Dealing with CORBA Exceptions in ORBWork

In ORBWork, a specific set of task manager errors has been introduced to deal with errors relating to interacting with the recovery framework. In the case of dealing with ORB-based errors, the task manager *catches* them directly and raises another `TaskManagerError` based on the context in which the ORB-error occurred. For example, in Figure 16, any CORBA exception returned by the `_bind` or the `Register` remote method invocation would be caught, translated, and *thrown* as a more meaningful `TaskManagerError_REGISTER` exception.

8.4 Task Manager Error Handling

The *rules* by which these errors are handled at runtime depends on the specific implementation of the METEOR$_2$ workflow model can be specified at *build-time* by the workflow designer.

The default *rules* for dealing with any task manager error in the `Execute()` (see 12) method is to log it, and let it and rethrow it to be handled by the `Activate()` method in a manner that is compliant with the workflow specification. However, it is possible to override the default error handling mechanism by specifying a *retry* option (with a `MAX_RETRIES` parameter) in a manner similar to task errors. This is specifically useful to handle the `TaskManager_TASK_EXECUTE` errors that might have been caused due to the unavailability of a processing entity.

If a task manager error is encountered in the `Activate()` method of the task manager, it would be detected in the `catch` block for `METEOR2_Error` types (as shown in the Figure 7). This would in effect lead to passing the resulting exception to the `Fail()` method (see figure 14) where it would be dealt with using *alternate* tasks.

Some task manager errors that cannot be resolved by the *rules* discussed above might result in adversely affecting the execution of the overall workflow process (e.g., *implementation specific errors* that are not handled by the workflow engine). Such task manager errors eventually result in a *workflow engine error* of type `WorkflowEngineError_SCHEDULING`. For example, in ORBWork, if a task manager is unable to *recover* (`TaskManagerError_RECOVER`) from its failed state, it is regarded as *corrupt* and therefore cannot be scheduled for usage thereby possibly affecting the normal flow of the workflow process. Another case in which the task manager is unsafe to be used is if the task manager is unable to *register* with the recovery components [Wor97] due to (say) a communication failure. This would result in a `TaskManagerError_REGISTRATION` error, which if not resolved through retries would result in `WorkflowEngineError_SCHEDULING` error.

Other task manager errors that cannot be resolved might not affect the overall execution of the workflow process, but could have future repercussions for the task manager itself. For example, errors resulting from logging task manager state, errors related to saving the state of the input and output data objects, etc. are all cases of these types of errors. Resolution of such errors is first attempted via *retrying* the operation a maximum of n number of times; if the problem persists, it is *reported* to the *workflow monitor* that should ideally be made available to the workflow administrator for monitoring the health of the workflow process enactment.

9 Workflow Engine Errors

This class of errors result from failures in either the scheduling, recovery or run-time administration units of the METEOR$_2$ workflow model (see figure 8). Some of these errors are fall-outs of task manager errors that could not be handled by the workflow engine (as discussed earlier).

Scheduling-related workflow engine errors include all errors that result from problems in enforcing inter-task dependencies between task managers during workflow enactment. Such an error is

represented as a `WorkflowEngineError_SCHEDULE` exception class in ORBWork. It is declared as a sub-classes of `WorkflowEngineError` object as:

```
// declare a specialization of WorkflowEngineError
class WorkflowEngineError_SCHEDULE : public WorkflowEngineError {
     // ...
}
```

The cause of scheduling errors could be multifarious. A workflow scheduler might not be able to activate a task manager due to a memory problem in the case of a centralized scheduler architecture [MSKW96]. On the other hand, in the case of a distributed workflow architecture, this could be caused due to communication failures.

Recovery-related workflow engine errors could affect the scheduler, task managers, administration units, internal data objects, and the recovery framework itself. These errors are defined as part of the workflow enactment, one for each component of the enactment system. For example, `WorkflowEngineError_RECOVER_LRM` represents an error in the workflow enactment engine resulting from the inability restore a *failed* local recovery manager.

In the case of errors relating to scheduling of task managers on particular hosts, users can specify alternate hosts on which alternate (or the same) task manager could be started. This redundancy makes it possible to mask task scheduling errors involving failed machines, or resource limitations on a particular machine. Errors that cannot be automatically handled by the WFMS are reported to a human via a workflow monitor with necessary details such as the cause of the error, the name of the reporting component, and the hostname of the machine where the error occurred.

Other workflow errors include errors caused by humans (e.g., turning a computer off) that could adversely affect the workflow process. These errors are reported (if detected by the workflow recovery framework) as *panic* messages to the workflow administrator thorough the workflow monitor. At this point, it would be the responsibility of the workflow administrator to fix the problem.

Just as in the case of *task* and *task manager errors*, there needs to be a *mapping* that would translate infrastructure related runtime errors into the `WorkflowEngineError` types that have been defined for a particular implementation of the METEOR$_2$ workflow model.

So far, we have not specifically discussed error handling for errors resulting due to loss of messages (e.g., a CORBA request). We have relied on the infrastructure (e.g., TCP/IP) to support these guarantees, although we realize that it might not be a safe assumption for most cases. A future version of ORBWork, could incorporate transactional messaging features using a transactional message queue or a TP-monitor.

10 Human-Assisted Error Handling

In ORBWork, it might not be possible to mask all errors in an automated fashion. In such cases, human assisted recovery is required to bring the WFMS to a consistent state. The *Workflow Monitor* (see figure 5) is a tool that is used as an administrative utility for critical errors that might need attention. Error messages that are generated during workflow enactment, are communicated to the monitor along with details regarding the whereabouts of the problem. The current version of the workflow monitor recognized three types of messages - status, alert and panic, depending on the nature of the situation being reported. Status error messages are reported during *normal* modes of execution, whereas alert and panic messages are reported during *abnormal* ones. For further details regarding the supported message-types and their semantics, refer to [Wor97].

11 Conclusion

In this paper, we have described the problem of workflow error handling in the context of the METEOR project. We first identified the requirements for dealing with errors in heterogeneous workflow environments. The current state-of-the-art in WFMS does not provide well-defined models to deal with the different types of errors that can occur during workflow enactment. In this paper, we have formulated an error handling approach for the METEOR$_2$ WFMS. This approach is unique to the area of workflow research since it provides the capability to partition workflow enactment errors in a unified manner based on the workflow model. Moreover, the error model that we have developed is based on the METEOR workflow model (task, task manager, workflow engine) rather than on infrastructure-, workflow architecture- or task-specific errors. This makes the error handling mechanism adaptable across different workflow infrastructures and engine architectures and is therefore applicable across different implementations of the METEOR$_2$ WFMS.

Our error handling mechanism is based on a top-down approach. It involves mapping heterogeneous infrastructure- and task-specific errors into an error model that we have defined for METEOR$_2$. Task-retries and alternate tasks that have been discussed in previous work in METEOR [KS95] have been integrated with the error model to provide a complete solution for dealing with errors encountered during workflow enactment. In addition, we have provided support for human-assisted error handling for situations that cannot be dealt automatically by the workflow enactment services. We have applied the error model and error handling mechanisms to ORBWork, the distributed workflow enactment service for the METEOR$_2$ WFMS.

In light of the overall problem of error handling and recovery in WFMSs, this paper provides a solution to dealing with errors that are *infrastructure specific* (e.g., CORBA, Web), *architecture specific* (e.g., ORBWork) and *model-specific* (e.g, METEOR workflow model) errors in a systematic manner. Our current error handing framework is limited to the context of workflow enactment services. Some of the areas that remain to be addressed in future work include providing better support for dealing with organizational errors (e.g., role-errors, security errors, etc) using dynamic and adaptive workflow systems. The error handling framework that we have presented in this paper provides a suitable infrastructure for developing higher-level error handling constructs in WFMSs.

Acknowledgements

This research was partially done under a cooperative agreement between the National Institute of Standards and Technology Advanced Technology Program (under the HIIT contract, number 70NANB5H1011) and the Healthcare Open System and Trials, Inc. consortium. See URL: *http://www.scra.org/hiit.html*. Additional partial support and donations are provided by Visigenic, Informix, Iona, Hewlett-Packard Labs, and Boeing.

References

[AAA+96] G. Alonso, D. Agrawal, A. El Abbadi, M. Kamath, and R. Gunthor. Advanced Transaction Models in Workflow Contexts. In *Proc. of 12th. IEEE Intl. Conference on Data Engineering*, pages 574–581, New Orleans, LA, February 1996.

[AAAM97] G. Alonso, D. Agrawal, A. El Abbadi, and C. Mohan. Functionalities and Limitations of Current Workflow Management Systems. Technical report, IBM Almaden Research Center, 1997. To appear in IEEE Expert, Special Issue on Cooperative Information Systems.

[ANRS92] M. Ansari, L. Ness, M. Rusinkiewicz, and A. Sheth. Using Flexible Transactions to Support Multi-system Telecommunication Applications. In *Proc. of the 18th Intl. Conference on Very Large Data Bases*, pages 65–76, Vancouver, Canada, August 1992.

[ASSR93] P. Attie, M. Singh, A. Sheth, and M. Rusinkiewicz. Specifying and Enforcing Intertask Dependencies. In *Proc. of the 19th Intl. Conference on Very Large Data Bases*, pages 134–145, Dublin, Ireland, 1993.

[CD96] Q. Chen and U. Dayal. A Transactional Nested Process Management System. In *Proc. of 12th. IEEE Intl. Conference on Data Engineering*, pages 566–573, New Orleans, LA, February 1996.

[Das97] S. Das. ORBWork: A Distributed CORBA-based Engine for the METEOR$_2$ Workflow Management System. Master's thesis, University of Georgia, Athens, GA, March 1997.

[DHL90] U. Dayal, M. Hsu, and R. Ladin. Organizing Long-Running Activities with Triggers and Transactions. In *Proc. of the ACM SIGMOD Conference on Management of Data*, 1990.

[DHL91] U. Dayal, M. Hsu, and R. Ladin. A Transactional Model for Long-running Activities. In *Proc. of the 17th. Intl. Conference on Very Large Data Bases*, pages 113–122, Barcelona, Spain, September 1991.

[Dog96] A. Dogac. Special-Theme Issue: Multidatabases. In *Journal of Database Management*. Idea Group Publishing, Harrisburg, PA, Winter 1996.

[EL96] J. Eder and W. Liebhart. Workflow Recovery. In *Proc. of the 1st. IFCIS Conference on Cooperative Information Systems*, Brussels, Belgium, June 1996.

[ELLR90] A. K. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz. A Multidatabase Transaction Model for InterBase. In *Proc. of the 16th. Intl. Conference on Very Large Data Bases*, pages 507–518, Brisbane, Australia, August 1990.

[Elm92] A. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[GHS95] D. Georgakopoulos, M. Hornick, and A. Sheth. An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, 3(2):119–154, April 1995.

[GMS87] H. Garcia-Molina and K. Salem. Sagas. In *Proc. of ACM SIGMOD Conference on Management of Data*, pages 249–259, San Francisco, CA, May 1987.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[Hol94] D. Hollingsworth. The Workflow Reference Model. Technical Report TC00-1003, Issue 1.1, The Workflow Management Coalition, Brussels, Belgium, November 1994.

[JK97] S. Jajodia and L. Kerschberg, editors. *Advanced Transaction Models and Architectures*. Kluwer Academic Publishers, 1997. To be published.

[KS95] N. Krishnakumar and A. Sheth. Managing Heterogeneous Multi-system Tasks to Support Enterprise-wide Operations. *Distributed and Parallel Databases*, 3(2):155–186, April 1995.

[Ley95] F. Leymann. Supporting Business Transactions via Partial Backward Recovery in Workflow Management Systems. In *GI-Fachtagung Datenbanken in Buro Technik und Wissenchaft*, Dresden, Germany, 1995. Springer-Verlag.

[Lin97] C. Lin. A Graphical Workflow Designer for the METEOR$_2$ Workflow Management System. Master's thesis, University of Georgia, Athens, GA, 1997. In preparation.

[Mos82] J. Moss. Nested Transactions and Reliable Distributed Computing. In *Proc. of the 2nd. Symposium on Reliability in Distributed Software and Database Systems*, pages 33–39, Pittsburgh, PA, July 1982. IEEE CS Press.

[MPS+97] J. Miller, D. Palaniswami, A. Sheth, K. Kochut, and H. Singh. WebWork: METEOR$_2$'s Web-based Workflow Management System. Technical Report UGA-CS-TR-97-002, University of Georgia, April 1997.

[MSKW96] J. A. Miller, A. P. Sheth, K. J. Kochut, and X. Wang. CORBA-based Run-Time Architectures for Workflow Management Systems. *[Dog96]*, 7(1):16–27, Winter 1996.

[OMG96] OMG. CORBA2.0/IIOP Specification. Technical report, Object Management Group, August 1996.

[Rat97] Rational. *UML Users Guide*. Rational Software Corporation, 1.0 edition, January 1997.

[RS95] M. Rusinkiewicz and A. Sheth. Specification and Execution of Transactional Workflows. In W. Kim, editor, *Modern Database Systems: The Object Model, Interoperability and Beyond*. ACM Press, New York, NY, 1995.

[RSW97] F. Ranno, S. Shrivastava, and S. Wheater. A system for specifying and coordinating the execution of reliable distributed applications. Technical report, The University of Newcastle upon Tyne, England, 1997.

[Saa95] H. Saastamoinen. *On the Handling of Exceptions in Information Systems*. PhD thesis, University of Jyvaskyla, 1995.

[SGJ+96] A. Sheth, D. Georgakopoulos, S. Joosten, M. Rusinkiewicz, W. Scacchi, J. Wileden, and A. Wolf. Report from the NSF Workshop on Workflow and Process Automation in Information Systems. Technical report, University of Georgia, UGA-CS-TR-96-003, July 1996. URL: http://LSDIS.cs.uga.edu/activities/NSF-workflow.

[SJ96] A. Sheth and S. Joosten. Workshop on Workflow Management: Research, Technology, Products, Applications and Experiences, August 1996.

[SJHB96] H. Schuster, S. Jablonski, P. Heinl, and C. Bussler. A General Framework for the Execution of Heterogeneous Programs in Workflow Management Systems . In *Proc. of the 1st. IFCS Intnl. Conference on Cooperative Information Systems*, Brussels, Belgium, June 1996.

[SKM+96] A. Sheth, K. J. Kochut, J. Miller, D. Worah, S. Das, C. Lin, D. Palaniswami, J. Lynch, and I. Shevchenko. Supporting State-Wide Immunization Tracking using Multi-Paradigm Workflow Technology. In *Proc. of the 22nd. Intnl. Conference on Very Large Data Bases*, Bombay, India, September 1996.

[Wan95] X. Wang. Implementation and Performance Evaluation of CORBA-Based Centralized Workflow Schedulers. Master's thesis, University of Georgia, August 1995.

[Wor97] D. Worah. Error Handling and Recovery for the ORBWork Workflow Enactment Service in METEOR. Master's thesis, University of Georgia, Athens, GA, May 1997.

[WS96] D. Worah and A. Sheth. What do Advanced Transaction Models Have to Offer for Workflows? In *Proc. of Intl. Workshop on Advanced Transaction Models and Architectures*, Goa, India, August 1996.

[WS97] D. Worah and A. Sheth. Transactions in Transactional Workflows. In *[JK97]*, chapter 1. Kluwer Academic Publishers, 1997.

[WWW97] The World Wide Web Consortium, 1997. URL: *http://www.w3.org/*.

[Zhe97] K. Zheng. Designing Workflow Processes in the METEOR$_2$ Workflow Management System. Master's thesis, University of Georgia, Athens, GA, 1997.

[ZNBB94] A. Zhang, M. Nodine, B. Bhargava, and O. Bukhres. Ensuring Relaxed Atomicity for Flexible Transactions in Multidatabase Systems. In *Proc. 1994 SIGMOD International Conference on Management of Data*, pages 67–78, 1994.