Handling Semantic Exceptions in the Large: A Multiagent Approach*

Sudhir K. Rustogi, Feng Wan, Jie Xing, and Munindar P. Singh
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-7534, USA

singh@ncsu.edu

Abstract

We consider *semantic* exceptions, which arise when a task yields results that are incorrect, inconsistent with related tasks, or incomplete. Semantic exceptions are especially prominent *in the large*, i.e., when we construct and execute a *workflow*. A workflow is a composite computation with several interoperating components and interacting processes. Detecting and resolving semantic exceptions is critical to the functioning of a workflow, especially when its member activities are autonomous, heterogeneous, long-lived, and interact in subtle ways. Unfortunately, present workflow techniques offer little support for exceptions. For modeling, they provide few abstractions beyond activity charts. For enactment, they are not flexible enough to allow a component to detect and resolve semantic exceptions properly.

We describe a multiagent approach for handling semantic exceptions. Our approach is based on high-level abstractions such as commitments, a process metamodel that accommodates commitments and allied concepts, a behavior model to specify agents, and an execution architecture that handles persistent and dynamic (re)execution. We can handle a variety of semantic exceptions by formulating a number of commitment patterns that cover the important situations. The behavior model and the commitment patterns are expressed as statecharts and executed in a rule-based system. In this way, advanced abstractions based on multiagent systems are mapped into conventional software techniques.

Topic: Models and paradigms

*This research has been supported by the U.S. National Science Foundation under grants IIS-9529179 and IIS-9624425, and by IBM Corporation.

1 Introduction

A *workflow* is a composite, long-lived activity that spans heterogeneous information systems and engages several humans carrying out a variety of tasks. Workflows are a natural model of computing in the large. Meaningful computations in large organizations or, more importantly, across enterprises are best modeled as workflows. This means that we need to be able to specify and manage workflows not only in traditional and virtual enterprises, but also in modern settings such as electronic commerce.

Current software technology provides only low-level abstractions and is woefully inadequate for the proper treatment of workflows. Abstractions such as remote methods and distributed transactions address only the simpler challenges in specifying and managing complex workflows. Specifically, remote methods offer no conceptual abstractions for composing activities, whereas transactions require a tighter coupling of activities than is appropriate. Even the extended transaction models [8] require the specification of compensatory methods or transactions, which are usually impossible to guarantee.

It is commonplace that specifying the routine cases of a workflow is easy. It is the exception handling that is difficult to specify. When the workflow is large and especially when it may involve different enterprises, then the exceptions can be more insidious and their correct handling correspondingly more important. In such settings, the participating entities are of heterogeneous designs and act autonomously. Present-day abstractions for complex activities are not applicable in such cases.

Because of the interest in workflows, scores of workflow tools exist, and a number of research approaches for workflows continue to be proposed. Typically, these suggest new ways of structuring activities that offer minor variations of how tasks may be ordered or whether they may be concurrently executed. Bluntly put, these are little more than variants of flowcharts, and have little to offer in terms of abstractions for handling exceptions. For instance, adding all exceptional flows would clutter up the chart beyond recognition. Some of these approaches are executed on rule-based systems, rather than as procedural scripts, but they are often conceptually modeled as flowcharts.

Exceptions and Interaction. We now briefly describe some of the relevant previous efforts that have guided our work and the intuitions that we have borrowed from them. We give a more elaborate literature survey in Section 5.

Exceptions have been studied within a number of communities: programming languages, software engineering, databases, and artificial intelligence. Miller & Tripathi define an exception as an "abnormal" computational state [18]. An abnormal state may result from an *error* (illegal), a *deviation* (legal, but undesirable), a *notification* (invalidating a previous assumption), or an *idiom* (legal, but rare). Miller & Tripathi then go on to study the case of errors. By contrast, we concentrate on the other three kinds of exceptions. Because such exceptions can be recovered from but must be handled in a manner that respects the application semantics, we term them the *semantic* exceptions.

Xu *et al.* address coordinated exception handling, which is essential in distributed systems [25]. Traditional programming language approaches assume that the detection and resolution of exceptions will follow the flow of control of normal execution and will happen within the scope of a single task. However, when there are interactions among components, additional kinds of exceptions may arise due to the interactions; further, their resolution will also be interactive. Our approach seeks to capture similar intuitions by defining classes of exceptions and their handling in terms of interactions, although the interactions in our case are captured in a high-level framework.

There is an increasing realization within several subareas of computing that proper models of computing and process must emphasize the notion of interaction, e.g., [23]. This view is related to the recent observation of the fundamental position of feedback in software systems and their evolution [17]. We take the view that agents are the most useful when they can interact with each other. Accordingly, we have been pursuing a research program of *Interaction-Oriented Programming (IOP)* to develop techniques for constructing multiagent systems (with suitably interacting agents) to carry out complex activities such as workflows. Here we present the component of IOP that deals with the agents' commitments to one another, which are crucial in their interactively detecting and resolving exceptions.

Exception Handling in Brief. Our multiagent approach for workflow modeling and enactment relies on two crucial properties of agents. First, the agents must interact at a high level, which is how they form and manage commitments to one another. These commitments are about the information the agents exchange, about changes to that information, and about each other's needs. For example, an agent would not only send results to others, but may commit to satisfying its consumers or to notifying its consumers if it ever modifies those results. Second, the agents must be persistent. Persistence is essential so the agents may form, manage, and act according to their commitments. For example, an agent may retry a task until the task produces results acceptable to it and to its peers. It would receive updates and send updated results to its consumers. If the agents did not outlast their tasks, such actions would be impossible.

The agents are guided by how they may form, manipulate, or revoke their commitments. We describe a small set of carefully engineered *commitment patterns* through which a given workflow can be structured. The commitment patterns capture the essence of a workflow beyond the activities that take place in it. In our approach, the patterns are translated into rules, which are assigned to different agents based on their roles, and executed. For reasons of space, we shall not discuss the implementation details.

Different kinds of exceptions can now be captured through sets of commitments patterns on part of the different agents. By executing according to their commitments, the agents can satisfy the high-level constraints of their workflow, which are stated only in terms of how the agents interface, not in terms of how the agents might be constructed. Interestingly, exceptions are handled in a purely interactive method. We now introduce our running example.

Example 1 This workflow involves four kinds of agents. A customer comes up a need to travel to a certain city on a certain date. She contacts her travel agent who in turn requests an airline and a hotel clerk to make appropriate reservations. The clerks are to send confirmations to the customer. The customer may have some additional requirements. For example, the hotel may not be close to the airport chosen by the airline clerk and for a late flight, that is an important constraint of the customer. If the customer's requirements are not met, she complains to her travel agent. He would then make a revised request to the clerks, let's say by trying to get an earlier flight. ▮

Although small, Example 1 is not trivial. A number of roles are involved, and communication among them does not follow a simple nesting of requests and responses. Even a task that executes successfully may need to be revisited. A number of semantic exceptions are possible: the airline or hotel clerk may fail to find a satisfactory itinerary. The customer may change her mind and request a different room or flight.

Contributions. We develop a novel approach that involves specifying, configuring, and executing a team of agents to robustly enact a workflow. Our approach includes a rich metamodel that captures the essential properties of a workflow through a small set of connectors and commitment patterns. Our execution framework enacts these specifications in a reentrant manner. Thus, our approach exploits the key features of the agent metaphor.

Our approach applies outside the realm of information systems. It provides a generic means of creating and managing teams of agents. The metamodel is a knowledge representation framework for complex, distributed activities. Crucially, we provide an operational characterization of the metamodel through the behavioral model for agents, which can then be realized in a rule-based system.

The key features of our approach, in which respects it improves over other approaches for exception handling, are the following.

- *Modularity.* The different components of the workflow are specified independently; the exceptions can be specified independently of each other. The tasks in a workflow are kept reusable; the workflow just wires them together as desired.

- *High-level abstractions.* Conventional approaches to workflow mostly consider only the activity abstractions. Some consider these directly in various low-level procedural representations; others in rule-based but still procedural representations. By contrast, we consider higher-level abstractions that seek to capture the meaning of the activities in an organizational setting. The specific abstractions we propose are motivated from the study of agents, but are realized in a conventional framework.

- *Autonomy.* Our approach maximizes the autonomy of the entities that own and execute the different tasks. The agents may represent different enterprises, and decide whether and when to perform any tasks. Thus autonomy is preserved during both normal execution and exception handling.

- *Persistence.* Our agents are persistent, and can carry out activities that are long-lived. This enables us to think of a workflow not as a single-shot execution of a work-case, but an ongoing process in which decisions made early on may be revisited as different kinds of exceptions occur.

- *Emphasis on interaction structure.* A lot of the effort in constructing a workflow is specific to the application, especially in terms of the reasoning through which the agents may try out the available alternatives. However, we concentrate on the key structural properties of the workflows, especially as they relate to its organizational structure.

Some of these properties are supported by other approaches as well, but we believe their combination and the specific techniques to realize them are unique to our approach.

Organization. Section 2 describes our workflow metamodel and presents its key concepts. Section 3 introduces our classification of semantic exceptions, showing how they are handled through combinations of the commitment patterns. Section 4 briefly describes our system architecture and implementation just to give the reader a flavor for how our ideas are realized. Section 5 discusses the relevant literature and important future directions.

2 Workflow Metamodel

A workflow management system (WFMS) controls the execution of workflows based on their specifications. Consequently, each WFMS implicitly comes with a *metamodel*, a language for specifying the workflows that can be managed by the WFMS. Specifying and executing a real-life workflow with current tools proves challenging for the following reasons:

- Heterogeneous information access and management is often necessary, but can be complex. We assume here that existing or upcoming approaches will be used to address this problem [20, 2].
- Current techniques generally assume a client-server architecture, sometimes with replication, although real applications can require true distribution. This problem too is being addressed elsewhere, e.g., [10, 19].
- Typical WFMSs are based on some kind of a flowchart metamodel, which is not much help in specifying complex workflows or in handling exceptions. A lot of effort goes into coding the procedures or scripts through which the workflow is executed. To a large extent, this is because of the inadequate treatment of exceptions. This is the problem that we address in this paper.

To succeed with our vision of using agents and commitments for enacting workflows and handling exceptions, we must develop a metamodel using which the WFMS can ensure that the agents interact properly, but without violating their autonomy or heterogeneity. We present a suitable metamodel, which is quite small and elegant, but includes some additional concepts and relationships. Figure 1 presents our metamodel in UML notation [9]. We discuss the key ingredients of this metamodel next.

Tasks. A task identifies a definite piece of work. Tasks may be atomic or compound. We model a task as taking inputs and producing results, both of which map to the resources (such as databases) that must be available. A task is executed only if it is triggered; upon completion it produces an event, which (through the connectors) can be used to trigger other tasks.

Capabilities. Capabilities implement tasks. Through an abstract interface, a capability defines the primary processing required for a task. The capabilities are polymorphic in that they may encapsulate more than one method interface to the underlying task. These interfaces are invoked based on which data are available to the agent. Upon failure of one method, the agent may attempt to invoke another method. For example, a customer may request an air reservation for a certain destination and dates. Later, she may wish to specify the airlines or class of service as well.

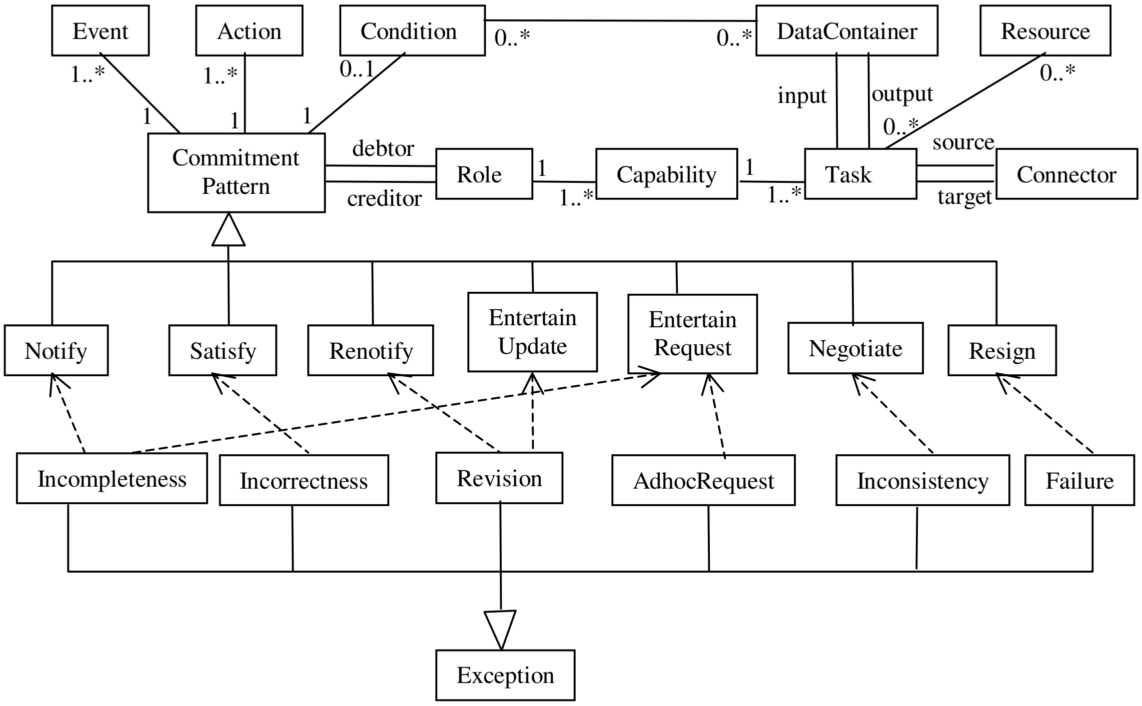

Figure 1: Workflow metamodel

Connectors. The connectors capture the control flow among the tasks in a workflow. Although the tasks themselves are reusable and hence independent of any application, the connectors wire them up into a particular workflow. Connectors also help capture the data flow among tasks by mapping the outputs of one to the inputs of the other. We allow a more or less standard set of connectors: Start, End, Linear, Fork, Branch, Or-join, and And-join.

However, their operational semantics goes beyond the traditional semantics. After the connector condition evaluates to true and the target tasks have been executed, there might be updates or corrections sent by the agents upstream. These would require the connectors to execute more than once, with different synchronization requirements. We term this behavior *reentrance*. Reentrance is essential to the execution of long-lived workflows by persistent agents.

Roles. A role is an abstract performer of tasks. A role must *provide* the capabilities required of any tasks assigned to it. A role may perform several tasks concurrently. During workflow execution, a concrete agent with matching capabilities is bound to each role. Importantly, roles are used to specify the commitments that apply in a workflow. Thus, roles capture the underlying organizational structure of the workflow.

Commitments. Abstractly, a commitment $\mathsf{C}(x, y, p, G)$ relates a debtor role x, a creditor role y, and a condition p, in the scope of a context group G. This means that x is obliged to y to satisfying p. The context group is the organization within which the workflow is executed. Having it as an explicit entity allows us to further control the evolution of the commitments—the details of how this is carried out are beyond the present paper. In our realization, the condition p usually involves rules or assertions in a rule-based language.

Unlike in databases, commitments are flexible, and can be revoked or modified. Almost always, however, the revocation or modification is constrained through *metacommitments*, which are commitments where the condition p itself involves commitments. In many cases, the commitment patterns apply in conjunction with the connectors, which carry the information about which commitments are made. However, the connection is not always direct and commitments may be specified even without an accompanying connector.

Agents. Agents are persistent active objects that can perceive, reason, act and communicate. Agents are autonomous and can volunteer to assume certain roles that would require them to perform certain tasks by executing their capabilities. Thus, an agent playing a role must implement all the capabilities that the role provides. When an agent adopts a role, it acquires the metacommitments of that role.

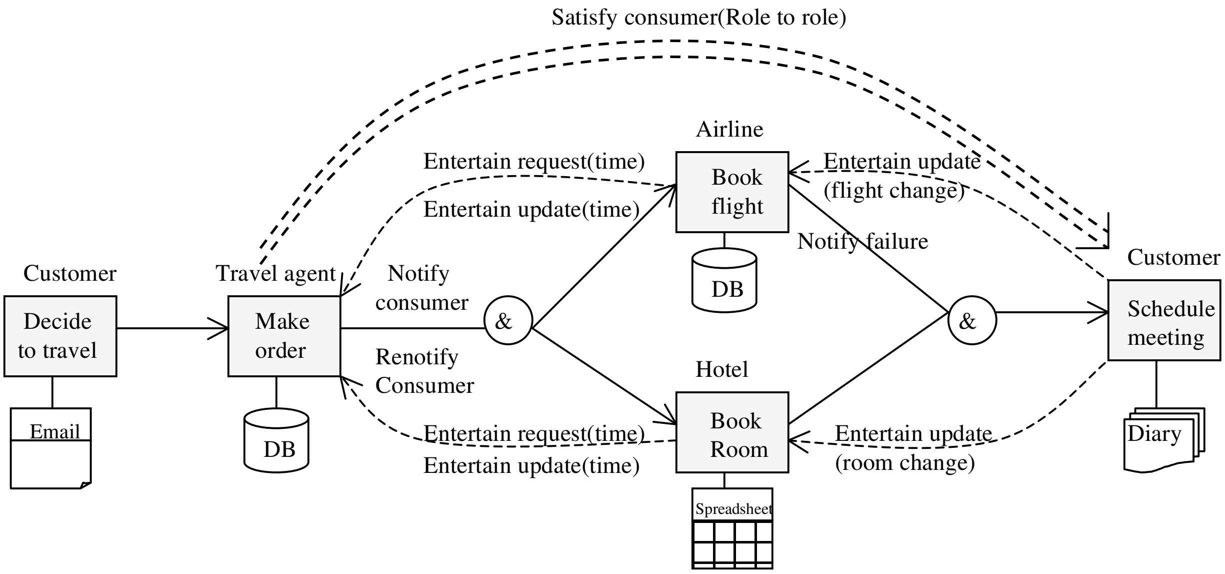

Figure 2: Workflow to plan a trip

Example 2 Figure 2 illustrates the metamodel applied on Example 1. The rectangles represent tasks to each of which a role is assigned; the customer role is assigned to two tasks. Each task has associated resources. The connectors capture the essential control and data flow. Along with each connector is a commitment from the source role to a target role. The single-dashed lines correspond to commitments against the orientation of the given connector; the double-dashed line corresponds to a commitment independent of the connectors. The commitments are that the roles will notify other roles, entertain requests and updates, and satisfy their customers. These commitments are precisely described below. ▮

2.1 Behavior Model for Agent

In order that appropriately minimal commitment patterns be specifiable, we require that the agents follow a general behavioral model. In other words, given a capability (realized in any manner that a vendor cares to use), we would like to wrap some structure around it. This structure has been designed exclusively to identify the states using which the agent

- becomes a persistent computation
- is able to enter into the different commitments

but without exposing any proprietary details of its design.

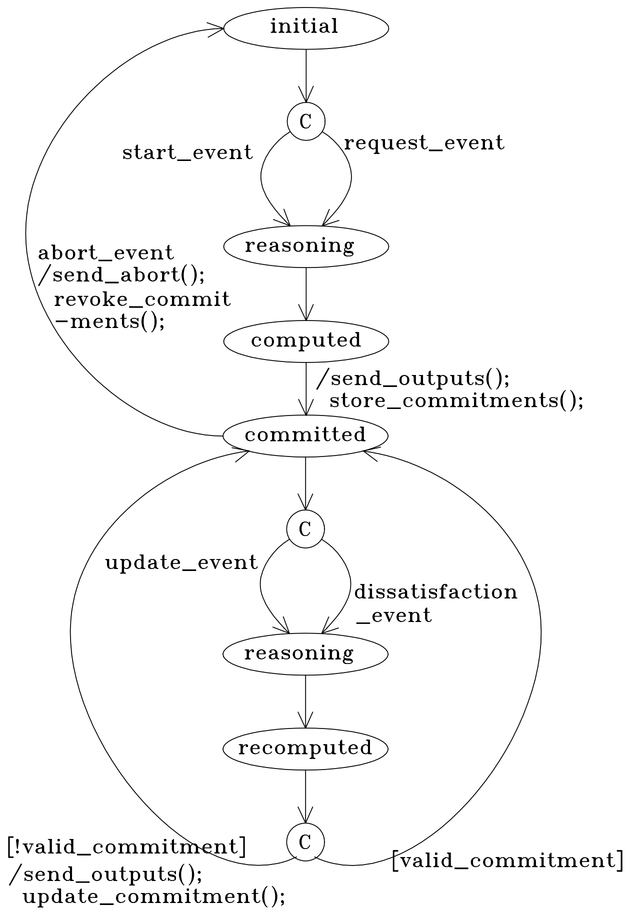

Figure 3: Behavioral model for commitments

Figure 3 shows the above behavioral model expressed as a statechart. Statecharts are well-established in software engineering as a means to specify concurrent computations [11]. Typically, such descriptions involve complex sequences of events, actions, conditions, and information flow that combine to form a system's overall behavior. A statechart is primarily composed of *states* (OR-states, AND-states, and basic states) and *transitions*. Transitions are labeled by an expression of the form $e[c]/a$. Intuitively, event e triggers the transition if condition c is true when e occurs. As a result, action a is performed. Each of e, c, and a is optional. The states in our statecharts are abstract and correspond to sets of physical states of the underlying computation that are considered equivalent.

Figure 3 describes the behavioral model for an agent who can commit. The agent begins in the *initial* state. On receiving a request or a control signal through a connector, it transitions to the *reasoning* state. When its computations conclude, it enters the *computed* state. In this state, the agent sends out its results to whoever it is dealing with—usually, the agent performing the next task in the workflow. As a general rule, whenever an agent sends out results it is committed to their accuracy.

Now further events (such as the receipt of an update or a dissatisfaction notification) may cause the agent to reexecute its reasoning. If the results previously declared by the agent change substantially, i.e., invalidate any commitments, then the agent can announce the new results. In that case, the agent commits to the new results instead of the old results. To keep the commitment patterns tractable and to simplify the execution of the workflows, it was important to keep the first execution and later executions separate.

The transitions of this behavior model are all optional in principle. The agent will have specific metacommitments that force it to carry out those steps.

2.2 Commitment Patterns

The commitments help the agents behave in a coherent manner to realize the robustness and flexibility that we desire. Specifying and managing commitments is at the heart of our approach. If we would go by just the formal definitions, we could make up an endless variety of commitments. But such arbitrary commitments would not be easy to understand, would not help structure the workflows properly, and would merely lapse into an ad hoc means for capturing inappropriate workflow designs. Indeed, constraining the specifications is the major purpose of metamodels.

For this reason, we define a small but expressively rich set of commitment patterns. By adding further structure to a workflow, these patterns help us specify workflows whose components truly do interact coherently. These patterns are based on a study of real-life workflows as well of examples from the literature. The patterns are *minimal*, in that each imposes the fewest reasonable restrictions. Importantly, the patterns can be *composed* so that they may be assigned in whatever combinations that make sense to the workflow designer.

Given the agents' behavioral model, the different commitment patterns are fragments of it that an agent is committed to obeying. The commitment patterns are converted into rules, which are treated on par with other rules and are executed by an agent adopting the given role. We presently have designed a dozen or so patterns. Of these, we describe some representative ones that apply directly in handling exceptions. Each of these is extracted from Figure 3.

Notify the consumer. This commitment pattern comes into effect when a role finishes its execution the first time. When it finishes execution for the first time, it enters the the *computed* state in the statechart of Figure 4. Operationally, the rule for this pattern is fired when the computed state is entered. This rules causes results to be sent to the given consumer and the associated commitment is created and stored.

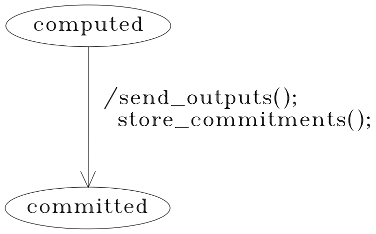

Figure 4: Notify the consumer (commitment pattern)

Renotify the consumer. This commitment pattern comes into effect when the agent has just completed its computation for the second or a later iteration, and some of its existing commitments are violated by the recently completed computations. The violation would typically because the predicate that the agent had committed has been falsified by the results just obtained. This pattern relates a producer to a consumer with respect to some data items. Under the pattern, the producer must renotify the consumer if the relevant data items have changed so as to violate the commitment. As usual, the new commitment to the consumer is created and stored. Note that the pattern allows a producer to treat different consumers differently.

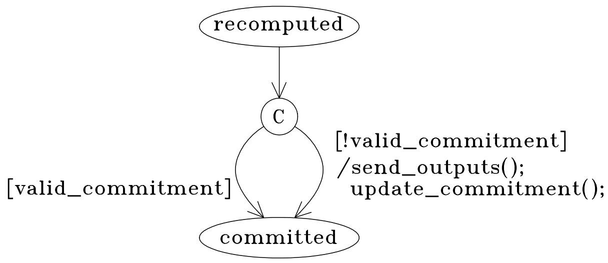

Figure 5: Renotify the consumer (commitment pattern)

Entertain request from another agent. This commitment pattern means that the agent should begin reasoning upon receiving a request from another agent. This pattern consists (from Figure 3) of the *start*, *reasoning*, and *computed* states, and the REQUEST_EVENT transition and the succeeding transition.

Entertain update from producer. This commitment pattern commits an agent to entertaining any revised data values that a producer may supply. If the consumer receives an update, it must reexecute its capability corresponding to that task. The specific method that is executed may be different from whatever the agent had executed previously, but the capability must still be executed. This pattern consists (from Figure 3) of the *committed*, *reasoning*, and *recomputed* states, and the UPDATE_EVENT transition and the succeeding transition. It is important to include this pattern, because in many cases, it would not be in the interest of the consumer to reexecute its capabilities. However, in many cases, it would serve the downstream agents or the workflow at large well if the capability was reexecuted.

Satisfy the consumer. This commitment pattern is similar to entertain-update pattern with UPDATE_EVENT replaced by DISSATISFACTION_EVENT. The practical ramifications of this pattern are much greater, however, because it enables an agent to report a complaint upstream. With the support of this pattern, the consumer can demand that a constraint it desires in its inputs be satisfied. Notice that the necessary constraint being violated is reported back to the producer so it can decide how to recompute the results so as to satisfy the consumer.

Negotiate Until Agreement. This commitment patterns applies whenever two or more agents are assigned to a composite role. The agents must produce one consistent set of results, to which end they must negotiate with each other. The specific negotiation techniques are not limited by our approach.

Resign. This commitment pattern recognizes that an agent may not always be able to succeed. Under this pattern, an agent tasked with something finally quits in failure. It is important to include this pattern so the whole workflow does not hang while an agent tries to carry out a task with which it cannot succeed.

3 Semantic Exceptions

As we indicated above, of the possible exceptions, our interest is in the *semantic*, i.e., application-level, exceptions. However, provided that a task does not end in error, the operating system, database management system, or programming language exceptions that it cannot transparently handle will become application-level exceptions.

Exceptions are classified in various ways in the literature. Two main categories are *anticipated* and *unanticipated* exceptions, respectively. The anticipated exceptions are easier to handle programatically; the others require some kind of autonomous, potentially intelligent, action by the software. Consequently, anticipated exceptions are conventionally treated through the specification of rules to handle them. Such exceptions are handled by the conventional database approaches. Unanticipated exceptions are treated only through user intervention or (at least in principle) through agents that can be hoped to be able to reason well enough to handle such exceptions.

In contrast to some previous approaches, our purpose here is not to specify the exact way in which the exceptions may be detected and handled, but how the agents enacting the given workflow may interact in order to felicitously handle the exceptions.

We define the following main types of exceptions. Our classification is more detailed than in the literature partly because we follow a richer model of workflows, which offers the potential for stating certain exceptions that traditional workflow metamodels cannot express, although they do occur in real life, anyway. We consider more detailed examples of each exception type and motivate our solution in terms of sets of commitment patterns.

Revision. This occurs when the inputs to an agent change after it has already declared its results or a request for new information is made. The agent who may have previously committed to some result may then have to ensure that its consumers too obtain the benefit of the new information. In such a case, the original producer may renotify the consumer thereby entering into a new commitment to the consumer. From the consumer's perspective, the receipt of such a notification would be an exception, because the consumer would already have processed the original version and forwarded any results to its consumers. The consumer should entertain this update.

Revisions are extremely common in most real applications of workflow technology, yet are not included in traditional metamodels. In our case, revision exceptions arise, because our framework makes the agents persistent even after the given instance of the workflow has been processed by them. If the agents are around, they can continue to observe their environment or communicate with other agents, either of which might cause them to change their mind and then to propagate their revisions further.

Our proposed approach to handling revisions is simply to require the producers to observe the renotify consumer pattern so they send out updates and require the consumers to observe the entertain update pattern.

Example 3 If the prospective traveler changes his mind and suggests a different date or destination city, he renotifies the travel agent about the revision on account of his commitment to renotify when a revision exception occurs. The revision exception is thus detected. It is completely handled when the travel agent entertains this update on account of his entertain update commitment. The travel agent, in turn, may propagate the effect of revision if (1) the effects are significant and lead to the violation of a prior commitment by the travel agent and (2) the travel agent is subject to the same commitment patterns. ▮

Incorrectness. This occurs when the inputs received by an agent from any source are simply incorrect. In this case, the incorrectness can be determined by the receiver without reference to inputs received from another source—this is what distinguishes incorrectness from inconsistency.

The inputs cause the consumer to repeatedly fail in its execution, thereby leading to its dissatisfaction. In this case, the producers must act to satisfy the consumer. Consequently, handling incorrectness exceptions involves the satisfy consumer pattern. Since a revised output is desired, we would also typically see the renotify consumer pattern and the entertain update pattern.

Example 4 If the airline agent cannot find a flight on the day of travel despite repeated trials (trying different airlines at different times of the day) he can either convey his failure to the customer or convey his dissatisfaction directly to the travel agent. In the case of failure, the customer, in turn, conveys her dissatisfaction to the travel agent. In both cases, when dissatisfaction is conveyed, a *reason* for the dissatisfaction is included. The reason states the constraints whose violation caused the dissatisfaction. Here, the reason is that the date of travel is not acceptable.

The travel agent must, therefore, provide a new date as he is bound by the commitment to satisfy his consumer. As the date was provided when the travel agent was requested to make the reservation, he propagates the dissatisfaction back (incidentally, also to the customer in this case).

The customer then changes the date. Once the date is changed the incorrectness exception has been handled. ∎

However, handling this exception can give rise to a new exception (revision) if modifications can be made. The revision exception is then taken care of as demonstrated in Example 3. Alternatively, if the modifications cannot be made, some other exception may be raised depending on the specific circumstances.

Incompleteness. This exception occurs when an agent receives fewer inputs than it needs to execute the appropriate capability. We emphasize that incompleteness cannot be removed in the design of the workflow. A common reason for the occurrence of this exception is that the capabilities may have several method signatures. Even if the inputs are sufficient for some of the signatures, they may not be sufficient for other signatures. Usually, under special circumstances, the agent will need to try signatures other than the simplest ones, for which purposes the inputs it received are incomplete. Handling incompleteness requires the commitment patterns of notify the consumer and usually also entertain requests.

Example 5 If the customer provides the date of travel and the city but not the state and if two or more cities across different states have the same name, the workflow enters a state of incompleteness exception. This is handled as the travel agent requests for more information (name of state) and the customer entertains the request and sort of completes the information by notifying his consumer (the travel agent). ∎

Inconsistency. This is when the outputs of two or more tasks are not mutually consistent. When the outputs are tied together because of some agreement, or desired agreement, among the agents producing them, there is usually a need for negotiation among those agents. Whenever two or more producers send outputs to their consumers, if their outputs conflict with respect to some stated constraints, the producers must negotiate among themselves before sending their outputs over. When the outputs are not tied together, the consumers are on their own to ensure that the inconsistency is eliminated. In this case, the consumers will rely upon handling schemes for incorrectness exceptions with some of the producers.

Consequently, the inconsistency exceptions can be handled through the negotiate until agreement pattern. As usual, applying this pattern may cause additional exceptions, which will then be handled appropriately.

Example 6 Suppose that an additional requirement specified by the traveler is that the hotel and airport be close together, say, within 5 miles of each other. However, if the airline clerk and the hotel clerk make reservations independently, they would not know the results of each other's executions. That is, the hotel booking agent would not know the location of the airport and the airline booking agent would not know the location of the hotel. Therefore, the locations selected by them could easily be in conflict with the customer's requirements. One way to handle this efficiently is to encapsulate the airline booking agent and the hotel booking agent together into a composite role. All roles within this higher-level role must satisfy the negotiate until agreement commitment pattern. ∎

Here too handling the exception may lead to other exceptions.

Exception class	Meaning	Patterns handled by
Revision	modified results	Renotify Entertain update
Inconsistency	conflicting set of values	Negotiate until agreement
Incorrectness	incorrect values	Satisfy
Incompleteness	missing values	Notify Entertain request
Ad hoc	unexpected request	Entertain request
Failure	no solution found	Resign

Table 1: Mapping exception classes to commitment patterns

Ad hoc request. A task (agent), during the persistent execution of a workflow discovers that it needs additional information to execute properly. It then issues that request to an agent who possesses the capability to supply that information. The requestee then supplies that information maybe by reexecuting appropriately. The challenge here is simply that the agent be able to follow a control path that diverges from whatever workflows in which it is engaged. Ad hoc requests can be handled through the entertain request pattern.

Example 7 The traveler may decide to check whether the given hotel has a shuttle service to the selected airport. Or, the travel agent may do so when suggesting that the traveler rent a car upon arrival. The hotel agent should be able to accommodate this request even though it is not part of the workflow as stated. ▮

Handling ad hoc requests could cause further exceptions that must then be handled according to appropriate patterns.

Failure. Sometimes, an agent can be forced to give up when the inputs are such that the underlying database system is unable to produce an acceptable answer.

If a consumer is repeatedly dissatisfied until it decides that it is not worth trying further, it can announce that the task it was attempting cannot succeed. The decision to declare failure may be based on all available method signatures having been attempted and resulted in failure (e.g., because the underlying database has been corrupted). The decision can also be based on the agent's knowledge of the importance of the task to the entire workflow. Much application-specific reasoning can go into this decision, but in the end when the agent has made the decision, we only show how the agent may announce it to its colleagues.

If the failure is along an OR branch, the rest of the workflow may survive. In the worst case, however, a failure can result in the termination of the entire workflow. The pattern to handle this exception is the resign pattern.

Example 8 If a traveler asks for a ticket below a certain price and only wishes to travel on a particular day, the airline agent may declare failure after trying the flights for that day and finding that all the cheap fares are sold out. ▮

Table 1 summarizes our results about how to map the exception classes to the commitment patterns. The lower part of Figure 1 shows the relationships pictorially through UML's dependencies.

4 Architecture and Implementation

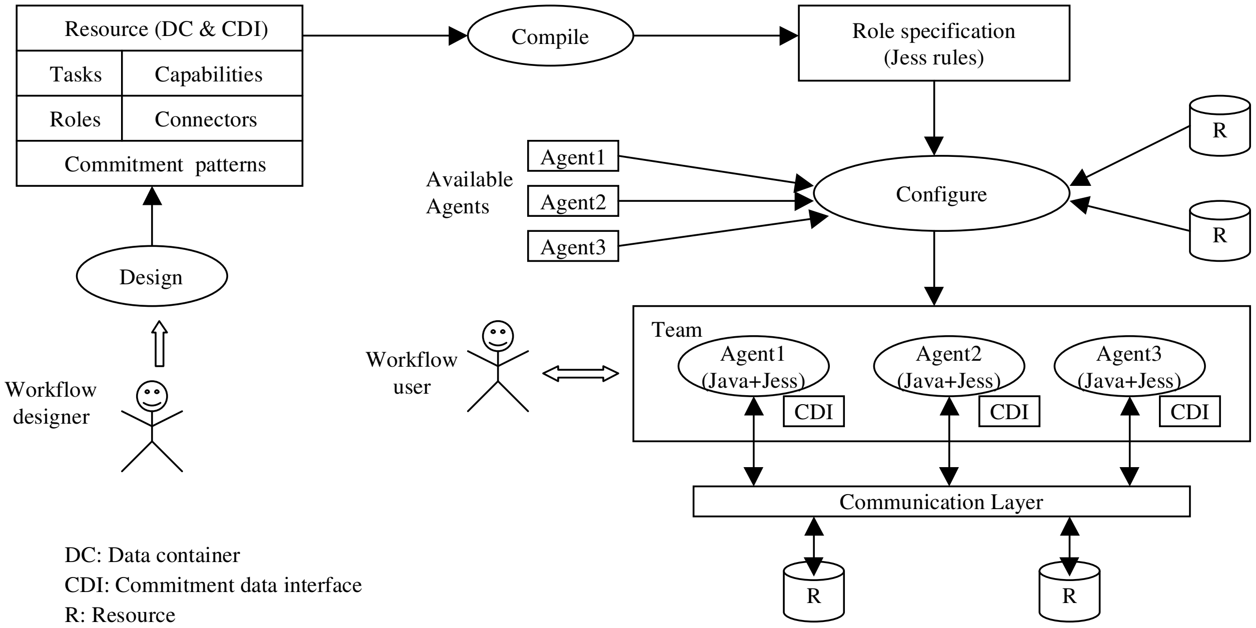

Figure 6: System architecture

Figure 6 shows our system architecture. Initially, a workflow specification is created. This is then compiled into a set of specifications for the roles.

Specification. Our metamodel translates naturally into a set of document type definitions (DTDs) for the extensible markup language (XML). XML is an emerging standard for exchange of information. We use our DTDs as an intermediate representation between a graphical interface and the reasoning system. We exploit XML tools, including an object interface for XML documents, to validate and parse workflow specifications.

Execution. Our execution framework is through a set of agents, each written in Jess, the Java Expert System Shell. Jess is a well-regarded, open source Java-based forward-chaining inference engine. A programmer can define rules and assert facts. Jess evaluates these rules and executes any actions embedded in them, such as asserting more facts or invoking the desired capabilities. Jess evaluates the rules against the facts, executing any rule all of whose antecedents are true, potentially asserting more facts, and executing still more rules until quiescense.

Jess offers seamless integration with Java: it can be invoked from or invoke Java method. This simplifies the implementation of a system such as ours. An agent is realized as a combination of Java and Jess. The Jess engine is embedded in a Java thread. The former reasons about the interactions with other agents while the latter facilitates these interactions by providing the means of accessing the external world.

Example 9 We now describe how our system executes on our running example. The traveler orders a trip from the travel agent, who requests bookings from the airline and the hotel (notify-consumer pattern). The airline and hotel clerks begin processing (entertain-request pattern). Both airline and hotel succeed and notify their consumer, the traveler. But the traveler is not satisfied, because the hotel is too far from the airport. She sends a dissatisfaction event to the travel agent along with the reason. The travel agent processes this dissatisfaction event (satisfy-consumer pattern), and then sends an update to the hotel (renotify-consumer pattern), but not to the airline, because the old commitment is still valid. The hotel clerk accepts the update (entertain-update pattern), and books the traveler at the airport location. He sends an updated confirmation to the traveler (renotify-consumer pattern). The traveler is satisfied this time and the workflow instance concludes. However, the different parties do not forget what they did in case there are further changes. ∎

5 Discussion

When constructing complex activities in large software environments, we must pay considerable attention to the exceptions that may occur therein. Workflows can add coherence to distributed activities in heterogeneous, open information environments. Workflows have a natural match with agents. To achieve progress in a dynamic, unpredictable world, however, the agents must be able to change their mind in a controlled way, handle deviations, and treat other rare events properly. To do so well, without complicating the normal specification, is the challenge we address here. Unfortunately, conventional workflow techniques are either rigid or unstructured, and therefore inapplicable for all but the simplest problems. The approach we develop introduces a number of ideas relating to agents for both modeling and enactment. Although our research was motivated by workflows, its results can apply wherever teams of agents may be employed to carry out complex activities in a coordinated manner.

5.1 Literature

Exception handling is the holy grail of workflow management if not of all software engineering. Too many approaches to exception handling have been proposed for us to review here. There are several products and a huge body of literature on workflows. However, the approaches fall into a small number of important categories. We describe representatives of these categories here.

Traditional. These include the approaches that are realized in current tools. Typically, these approaches include a simplistic flowchart-like modeling language along with a centralized execution engine that steps through the chart by invoking the activities in turn as their predecessors are completed. These approaches are notoriously inept at exception handling, because they demand excessive connections among the tasks to cover all the cases. Although these approaches have improved as a group, they still have not improved enough [5].

Database. These approaches provide additional functionality for structuring and executing activities. The enhancements are through the use of some kind of extended transaction models (ETMs). ETMs can be used to describe what to do in the case of a component activity of the workflow fails.

The actions are typically along the lines of retrying the activity that failed, undoing the activities that previously succeeded (i.e., committed), or undoing some of the activities and then retrying the rest of the workflow.

The Carnot Project developed a generic event scheduling approach based on branching-time temporal logic [1]. This approach captures the skeletal behavior of individual tasks and transactions through significant events based on ACTA [6]. Singh developed a related approach based on guard evaluations, which supported distributed execution [19]. This approach has also been reimplemented in a more conventional distributed object framework in METUFlow [10]. These approaches allow flexible scheduling, which is essential to handling deviations in workflows.

The CREW approach provides one of the best recent approaches from the database community [14]. Kamath & Ramamritham develop a rich model for workflow that includes failure handling and enables *opportunistic* rollback, restricting the extent of the compensation activities. They consider the interactions among the different tasks. Kamath & Ramamritham can accommodate interactions among workflows, including the important case where a workflow feeds information to another, and can change its results under some circumstances.

Chiu *et al.* present a number of exception resolution techniques used in the ADOME-WFMS project. For each task, they specify whether it is optional, critical, repeatable, or replaceable. Exceptions can then be handled by maintaining the workflow but modifying some of the capabilities or task assignments, modifying the workflow structurally for this case (e.g., by skipping a task), or evolving the workflow to obtain an alternative model. The exception handling can exploit the composition hierarchy of tasks in the model.

Conversation-based. The *language for action* metamodel involves commitments as well [24], and is applied in the ActionWorkflow tool. This metamodel uses *loops* representing a four-step exchange between a *customer* and a *performer*: (a) a request from the customer, (b) negotiation by the two about the task, (c) actual performance of the task, and (d) evaluation of the performance by the customer. A step may potentially be nested with other loops.

Agent-based. The connection of agents to workflows has long been recognized. Agents naturally can carry out heterogeneous activities and coordinate their efforts with each others. The agents can negotiate with each other and provide the reasoning with which to diagnose and handle exceptions and other failures.

Klein exploits a knowledge base of generic exception detection, diagnosis, and resolution expertise [16]. Specialized agents are dedicated to exception handling. This approach is complementary to ours, and the two approaches can be combined in two ways. One, we can use special roles designed only to detect and diagnose exceptions. Two, we can include in each agent a certain reasoning capability for exception detection and diagnosis. Either of these approaches. would be important for unanticipated exceptions.

The advanced decision environment for decision tasks (ADEPT) project also considered workflow management [13]. This project emphasized negotiation among agents. However, the underlying notion of commitments does not allow contextual nesting, as in our approach.

Operations support. In their SmartObjects project, Vaishnavi *et al.* adapt artificial intelligence techniques for operations support, which involves interaction among computational tasks and humans [22]. For our purposes, Vaishnavi *et al.* emphasize the bidirectional, proactive nature of the interactions among the components, which cannot be easily expressed in simplistic client-server style procedure calls. Therefore, they explicitly represent the control knowledge and the structural properties of their interactions. We carry this further through the use of commitments to capture the structural relationships in an organizational sense.

Rule-based. We separate out the rule-based approaches, although they include both database and agent-based approaches. A number of approaches use rules to capture the control flow of a workflow. Rules provide a higher-level of description of the actions that must be taken, by an active database, external activity manager, or the agents who execute the workflow. Sometimes the rule-based approaches combine in representations and reasoning to keep track of how they have executed so they may be undone or retried if appropriate.

The Carnot Project was one of the first to apply agents to workflow management. Carnot used a small set of agents to cooperatively carry a workflow. The agents represented the desired workflow as an instance of a transaction model. They used rules to enact the workflow and to detect exceptions. The rule system supported some nonmonotonic reasoning, which was used to handle exceptions—first the system could execute whichever tasks were enabled; a later exception would remove the justification for doing the task and create a trigger to begin the compensating activity. This system was evaluated on a telecommunications service provisioning workflow [21].

More recently, the WIDE project also uses rules extensively [4]. The WIDE project also uses rules, but separates out the rules for exception handling. WIDE considers the case of deviations from workflows, but leading up to workflow evolution. Casati considers three main alternatives: patch the workflow external to the model, modify the workflow specification to handle the present deviation, evolve the workflow [5]. The first option offers no support for the user and the second requires too much work for one case, so Casati prefers the last. However, evolving a workflow is not a simple task. By contrast, our approach although using rules like WIDE, captures the commitments explicitly, which means that handling deviations is simplified.

Kappel *et al.* propose templates to capture these rules in TRIG$_{flow}$ [15]. Some templates are geared toward synchronization, but deal with the low-level aspects of synchronization.

Process modeling. This includes the relevant work from the software engineering community. Borgida & Murata study ad hoc deviations from workflows. They capture the allowed deviations through workflow models [3]. They define exceptions as constraint violations. Borgida & Murata represent workflows as objects to enable reasoning about them and to handle deviations.

The PROSYT project studies deviations in the context of process support systems, especially those involving several human tasks [7]. Cugola observes that traditional process models are based on activities and proposes an alternative approach based on *artifacts*—the objects produced by the process, which in his case are things like folders and repositories, using which humans work collaboratively. Cugola allows deviation handling policies, which result in the invocation of an action such as abort or continue with options to inform the user. However, the reconciliation activities are explicitly left outside the scope of this approach.

Our approach. We now briefly reflect on how our approach differs from and builds on the literature.

Unlike the temporal logic work in Carnot and by others, our present approach considers the higher-level abstractions of commitment patterns explicitly. Like CREW, we consider higher-level abstractions, but unlike CREW, we also consider abstractions that relate to the organizational aspects of the workflow. Some of our patterns are designed to accommodate opportunism in rollback, e.g., by renotifying consumers of updates only when necessary.

Like ADOME-WFMS, our approach captures the different properties of each task in the workflow. We can handle exceptions by attempting different methods from a capability interface and by modifying the structural properties of the workflow. The agents handle exceptions cooperatively with each other in a way that relies upon their organizational relationships and the structure of their tasks.

Like the language for action approach, we give primacy to the agents' commitments. However, the former metamodel has some limitations. It only considers two actors at a time, and does not explicitly consider the surrounding organizational structure. It cannot easily accommodate modifications or revocations of the commitments. All of these problems are avoided in our approach. Our approach fits squarely within the agents approaches, but emphasizes the decentralization and interactions of the agents. Intuitions related to operations support are also captured, because the agents can assist humans in carrying out complex tasks, doing the routine workflow and anticipated exceptions, but providing the users structural support for any exceptions they have to manually handle.

Rules have a natural fit with flexible execution and exception handling. However, most existing work on rules has not progressed to study the higher-level patterns involving rules, especially when we are interested in long-lived workflows and seek to capture how the autonomous entities carrying out the workflow are organized. These high-level patterns, even beyond TRIG$_{flow}$, can be derived from commitment patterns and provide a high-level, yet operational account of exception handling.

Our metamodel includes a basic fabric of activities carried out by agents. However, on this fabric, we build a structure of commitments involving the artifacts produced by the agents. Although, like PROSYT, we use artifacts, we can capture a far richer semantic through the commitments among the agents. The commitments also capture enough of the interactions of the different participants that if the participants have the necessary domain knowledge, the commitments help them reconcile the different activities to help achieve coherence among their activities.

5.2 Directions

A number of interesting technical problems are opened up by our research. One of the charms of our approach is that it synthesizes conventional software engineering techniques (namely, statecharts and process modeling) with AI techniques (namely, agents and commitments) to develop a powerful approach for workflow management. Each of these ingredients can be further improved. Specifically, on the conventional side, we are investigating a formal semantics for our approach that goes beyond the conventional statechart semantics in terms of allowing reentrance and revision. On the AI side, we are investigating enhanced representations for teams and organizations that would better accommodate the challenges of coherent, long-lived, complex computational activities.

References

[1] Paul C. Attie, Munindar P. Singh, E. Allen Emerson, Amit Sheth, and Marek Rusinkiewicz. Scheduling workflows by enforcing intertask dependencies. *Distributed Systems Engineering Journal*, 3(4):222–238, December 1996.

[2] R. Bayardo, W. Bohrer, R. Brice, A. Chichocki, G. Fowler, A. Helal, V. Kashyap, T. Ksiezyk, G. Martin, M. Nodine, M. Rashid, M. Rusinkiewicz, R. Shea, C. Unnikrishnan, A. Unruh, and D. Woelk. InfoSleuth: Semantic integration of information in open and dynamic environments. In *[12]*, pages 205–216. 1998. (Reprinted from *Proceedings of the ACM SIGMOD Conference, 1997*).

[3] Alex Borgida and Takahiro Murata. Workflows as persistent objects with persistent exceptions - a framework for flexibility. In *CSCW Workshop: Towards Adaptive Workflow Systems*, Seattle, WA, 1998. At ccs.mit.edu/klein/cscw98/.

[4] F. Casati, P. Grefen, B. Pernici, G. Pozzi, and G. Snchez. Wide workflow model and architecture. Technical Report 96.050, Dipartimento di Elettronica e Informazione, Politecnico di Milano, 1996.

[5] Fabio Casati. A discussion on approaches to handling exceptions in workflows. In *CSCW Workshop: Towards Adaptive Workflow Systems*, Seattle, WA, 1998. At ccs.mit.edu/klein/cscw98/.

[6] Panos K. Chrysanthis and Krithi Ramamritham. Synthesis of extended transaction models using ACTA. *ACM Transactions on Database Systems*, 19(3):450–491, September 1994.

[7] Gianpaolo Cugola. Tolerating deviations in process support systems via flexible enactment of process models. *IEEE Transactions on Software Engineering*, 1999. Special issue on Managing Inconsistency in Software Development. To appear.

[8] Ahmed K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, San Mateo, 1992.

[9] Martin Fowler. *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley, Reading, MA, 1997.

[10] Esin Gokkoca, Mehmet Altinel, Ibrahim Cingil, E. Nesime Tatbul, Pinar Koksal, and Asuman Dogac. Design and implementation of a distributed workflow enactment service. In *Proceedings of the International Conference on Cooperative Information Systems (CoopIS)*, pages 89–98, 1997.

[11] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[12] Michael N. Huhns and Munindar P. Singh, editors. *Readings in Agents*. Morgan Kaufmann, San Francisco, 1998.

[13] N. R. Jennings, P. Faratin, M. J. Johnson, T. J. Norman, P. O'Brien, and M. E. Wiegand. Agent-based business process management. *International Journal of Cooperative Information Systems*, 5(2&3):105–130, 1996.

[14] Mohan Kamath and Krithi Ramamritham. Failure handling and coordinated execution of concurrent workflows. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 334–341, 1998.

[15] G. Kappel, S. Rausch-Schott, W. Retschitzegger, and M. Sakkinen. Rule patterns - bottom-up design of active object-oriented databases. *Communications of the ACM (CACM)*, 42, 1999. To appear.

[16] Mark Klein. Exception handling in agent systems. In *Proceedings of the Third International Conference on Autonomous Agents*, Seattle, Washington, 1999.

[17] Meir M. Lehman, Dewayne E. Perry, and Wladyslaw M. Turski. Why is it so hard to find feedback control in software processes? In *Proceedings of the 19th Australasian Computer Science Conference*, 1996.

[18] Robert Miller and Anand Tripathi. Issues with exception handling in object-oriented systems. In *European Conference on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 85–103, 1997.

[19] Munindar P. Singh. Synthesizing distributed constrained events from transactional workflow specifications. In *Proceedings of the 12th International Conference on Data Engineering (ICDE)*, pages 616–623, February 1996.

[20] Munindar P. Singh, Philip E. Cannata, Michael N. Huhns, Nigel Jacobs, Tomasz Ksiezyk, Kayliang Ong, Amit P. Sheth, Christine Tomlinson, and Darrell Woelk. The Carnot heterogeneous database project: Implemented applications. *Distributed and Parallel Databases: An International Journal*, 5(2):207–225, April 1997.

[21] Munindar P. Singh and Michael N. Huhns. Automating workflows for service provisioning: Integrating AI and database technologies. *IEEE Expert*, 9(5):19–23, October 1994.

[22] Vijay K. Vaishnavi, Gary C. Buchanan, and William L. Kuechler, Jr. A data/knowledge paradigm for the modeling and design of operations support systems. *IEEE Transactions on Knowledge and Data Engineering*, 9(2):275–291, April 1997.

[23] Peter Wegner. Why interaction is more powerful than algorithms. *Communications of the ACM*, 40(5):80–91, May 1997.

[24] Terry Winograd and Fernando Flores. *Understanding Computers and Cognition: A New Foundation for Design*. Addison-Wesley, Reading, MA, 1987.

[25] J. Xu, A. Romanovsky, and B. Randell. Coordinated exception handling in distributed object systems: from model to system implementation. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS-18)*, 1998.