Multiagent Workflow Management

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

Abstract

Workflows are composite computations that involve a number of tasks, usually executed on diverse information resources. Workflows arise in virtually all applications of computing. Although workflows are important, present workflow technology is quite poor. For programming, it offers few abstractions beyond activity charts. During enactment, it cannot handle exceptions and revisions properly, both of which are important concerns where workflows are useful. Previous AI-based techniques have been better, but they too lack the high-level abstractions for engineering workflows. We develop a multiagent architecture for workflow management. This requires abstractions such as commitments, a metamodel that accommodates commitments and allied concepts, a behavior model to specify agents, and an execution architecture that handles persistent and dynamic (re)execution.

1 Introduction

A *workflow* is a composite, long-lived activity that spans across heterogeneous information systems and engages several humans carrying out a variety of tasks. Traditional approaches to computing have led to islands of automation. With the expansion of computing and communications, users increasingly expect that related information systems will interact properly with each other. 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, based on distributed computing and databases and dealing exclusively with low-level abstractions, is woefully inadequate for the proper treatment of workflows. This is because the associated abstractions—remote methods and distributed transactions—address only the simpler challenges in integrating heterogeneous systems. 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 require the specification of compensatory methods or transactions, which are usually impossible to guarantee.

Two major challenges are not properly addressed by any existing approach:

- *Exceptions.* Because workflows exist in heterogeneous environments, their execution frequently runs into exceptions, which are notoriously difficult to handle.
- *Revisions.* Because workflows are long-lived, potentially lasting forever, they must release their results prematurely. This means that results ought to be able to be revised.

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 by specifying how tasks may be ordered or whether they may be concurrently executed. Bluntly put, these are little more than flowcharts, and have little to offer in terms of abstractions for addressing the above challenges. For instance, adding all exceptional flows would clutter up the chart beyond recognition. Some of these approaches are executed on rule systems, rather than procedurally, but they are often conceptually modeled as flowcharts.

The importance of workflows to computing and the poverty of existing approaches suggests a great opportunity for AI. We discuss several AI approaches below.

Approach in Brief. We apply agents to workflow management in a way that naturally addresses the above challenges. We rely 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 they 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. This is essential, so the agents may form, manage, and act according to their commitments. For example, an agent may retry a task until it produces results acceptable to it and to its peers. It would receive updates and send updated results to its consumers. If the agents didn't outlast their tasks, such actions would be impossible.

The agents reason about the formation and manipulation (including revocation) of social commitments. This reasoning is constrained through specified *metacommitments*. Although, in principle, an infinite variety of metacommitments can be defined, it would not be conceptually appropriate to allow all of them. Accordingly, we limit this variety through a small set of carefully engineered *commitment patterns* through which a given workflow can be structured. The commitment patterns are translated into a set of rules, which are assigned to different agents based on their roles.

The commitment patterns are conceptualized as statecharts and can be directly composed. These statecharts are based on a single uniform template, which specifies the general behavior model for all our agents. This model abstracts out the main capabilities that agents must have to carry out their assigned tasks. It includes states denoting important stages in the life of an agent that can explicitly reason about its commitments to continually entertain new inputs and reconsider its results. Transitions among these states capture the specific commitment patterns that apply.

The agents must have memory of their commitments to others and others' commitments to them. This might seem to require significant storage and overhead, but in real-life settings, the situation is much better. This is because wherever a workflow is carried out, even with current approaches, some record is kept of the commitments. For example, in a university registration system, most of the data reflects commitments by the university to the students or by the faculty to the university. In an airline reservation system too, most data is about commitments. Consequently, our architecture includes a *commitment data interface*, using which the agents can retrieve and modify their commitments from some underlying database. The commitments themselves, therefore, add no storage overhead; the agents only additionally represent the applicable commitment patterns.

The workflow is specified generically in terms of the roles that would enact it. The agents volunteer to play different roles. However, they may play a role only if (1) they have the capabilities to perform the tasks required of that role and (2) are willing to adopt the commitments specified for that role. The assignment of agents to roles happens dynamically. An agent may play more than one role. The roles can be nested when the agents must agree on their solutions in order to satisfy a mutual customer.

Running Example. 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 traveler. 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 for 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.

Contributions. To our application area, we contribute a novel approach that involves specifying, configuring, and executing a team of agents to 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. It is important that we provide an operational characterization of it through the behavioral model for agents and its realization in a rule-based system.

Organization. Section 2 describes our workflow metamodel and presents its key concepts. Section 3 introduces an operational characterization of the commitment patterns, and Section 4 shows how they can be mapped to a rule-based execution framework. Section 5 gives our system architecture and discusses our implementation. Section 6 discusses the literature and 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 workflows that can be managed by the WFMS. Typical WFMSs are based on some kind of a flowchart metamodel, which isn't much help in handling the complex situations where workflow technology would be the most beneficial. A lot of the effort goes into coding the procedures or scripts through which the workflow is executed. While some of the complexity of the scripts arises from accessing legacy data, much of it is because of the ad hoc manner in which exceptions in the workflow are handled. Because the present research deals with the challenges of exceptions

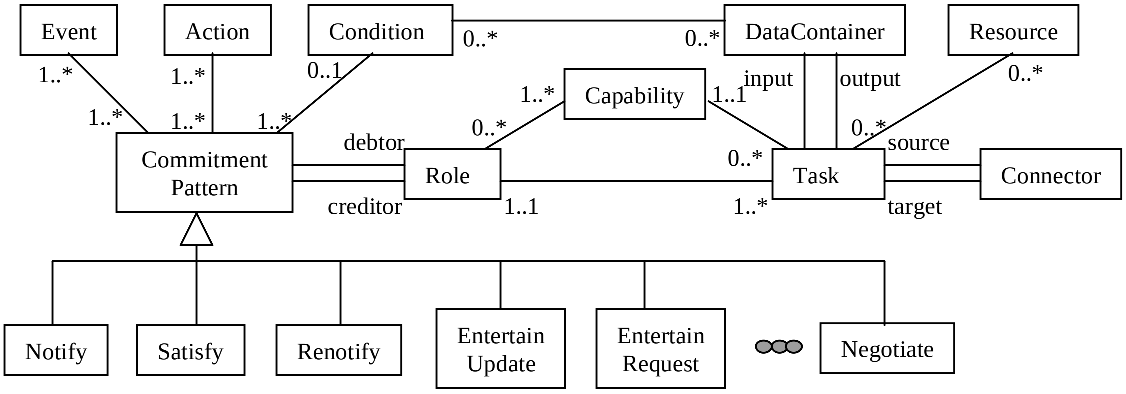

Figure 1: Workflow metamodel

and revisions, we assume that data access can be supported through techniques developed by others [Bayardo *et al.*, 1997].

However, our use of agents and commitments entails that we develop a metamodel using which the WFMS can ensure that the agents interact properly, but without violating their autonomy or heterogeneity. This metamodel is quite small and elegant, but includes some additional concepts and relationships. Figure 1 presents our metamodel in UML notation [Fowler, 1997]. 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 of a task.

Connectors

The connectors capture the control flow among the tasks in a workflow. Connectors also provide the context in which data flow among tasks may be captured by mapping the outputs of one to the inputs of the other. A connector may have multiple inputs and outputs. 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 by allowing reentrance with or without synchronization (Section 4.1).

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.

Commitments

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 are beyond the present paper. The condition p may involve relevant predicate. Unlike in databases, these commitments are flexible. They can be revoked or modified. Almost always, 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 entities 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.

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

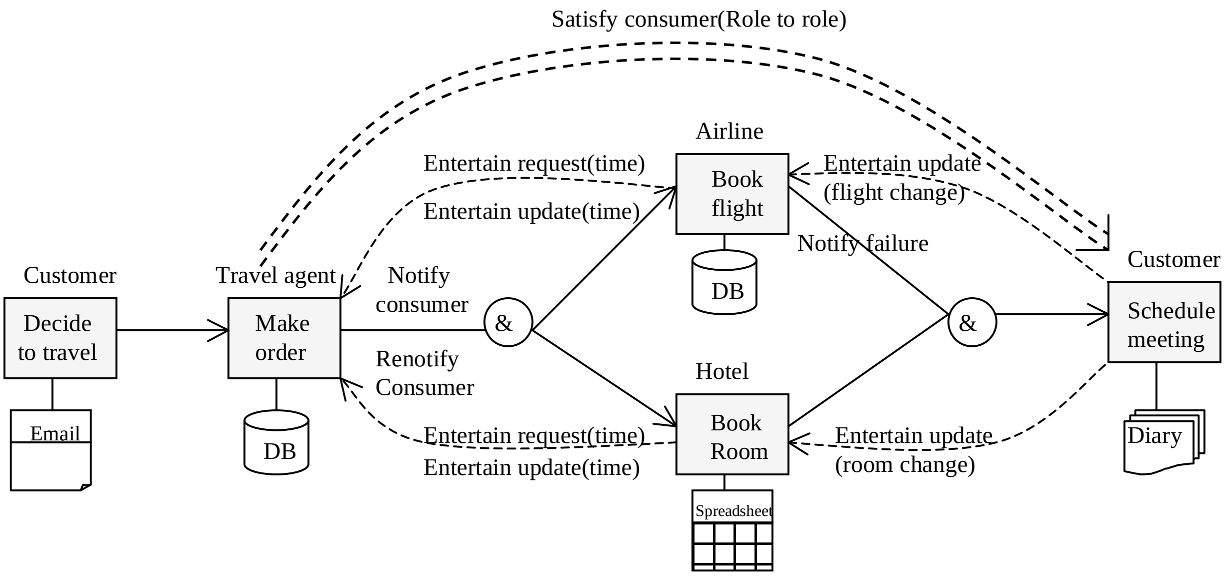

Figure 2: Workflow to plan a trip

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. ▮

3 Commitment Patterns Operationally

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 just go by the formal definitions, we could make up an endless variety of commitments. But such arbitrary commitments would not be easy to understand and would indeed lapse into an ad hoc means for capturing inappropriate workflow designs. Indeed, constraining the specifications is the major purpose of metamodeling.

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.

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 without exposing any proprietary details of its design.

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 [Harel, 1987]. 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. On receiving a request or a control signal, the agent begins reasoning. At the end, it releases and commits to its results. Further events may cause the agent to reexecute its reasoning. If its results change substantially, i.e., invalidate any commitments, it announces the new results and commits to them. The steps of this model are all optional in principle. The agent will have specific metacommitments that force it to carry out those steps.

Figure 4 shows how an agent reasons. Much of the complexity of this figure arises from allowing nested agents, i.e., agents that are teams. The member agents presuppose or guess some inputs, perform some temporary computations, and negotiate with each other. The process iterates until all members of a team agree, in

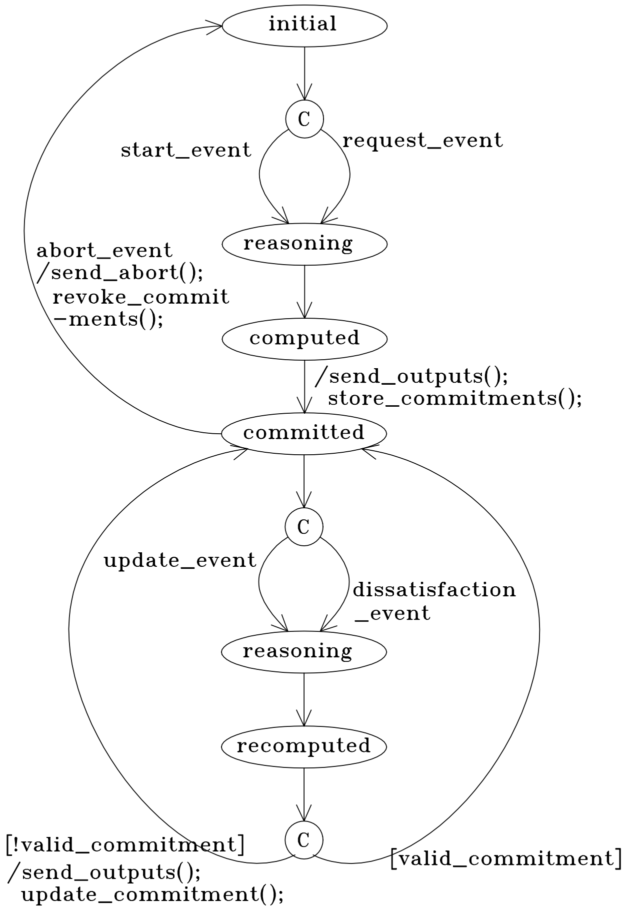

Figure 3: Behavioral model for commitments

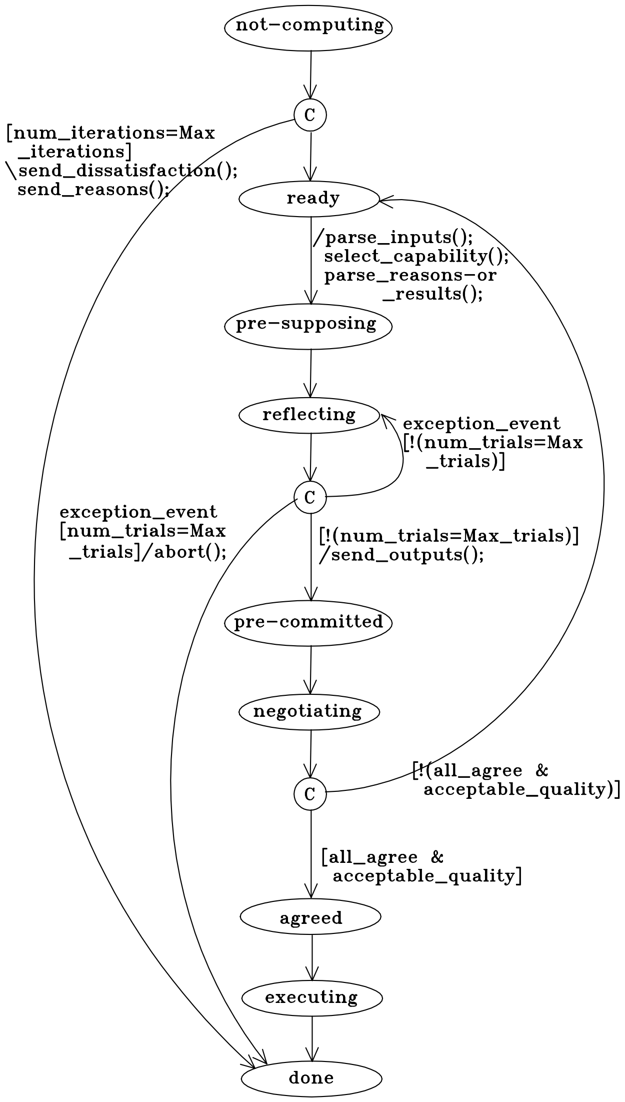

Figure 4: Model for reasoning (and negotiation)

which case the agreed upon results are made permanent within the team. If they fail to agree, they send a failure notification to their consumer. The number of times a team may be made to reason on the same problem is bounded; exceeding this bound results in dissatisfaction to its requesters.

Given this behavioral model, the different commitment patterns are fragments of it that an agent is committed to obeying. We presently have a dozen or so patterns. Of these we describe some representative ones that apply on our running example. Each of these is extracted from Figure 3.

4 Rule-Based Execution

Figure 5 describes the generic execution model followed by our agents. The agents are rule-based, but can execute procedural capabilities. Jess is the rule-based engine we use (see Section 5).

In obtaining a rule-based characterization of multiagent teams for workflow, we have to bring out some additional details of the execution framework. The rules we develop are based on the following.

- specific instance of the workflow
- tasks in the workflow
- events dealing with the capabilities that implement the tasks
- states of the agent in performing a task
- what iteration of the given task is being attempted
- states of the agent in interacting with others.

There are two main ways to implement the connectors. One, a designated agent would handle a connector by receiving "done" messages from the sources and sending "ready" messages to the targets. The target agent can then decide whether to wait for additional conditions. Two, there is no designated agent for a connector. Each source sends "done" messages to each target of a given connector. The target agent must check the connector conditions and its own guard conditions to decide

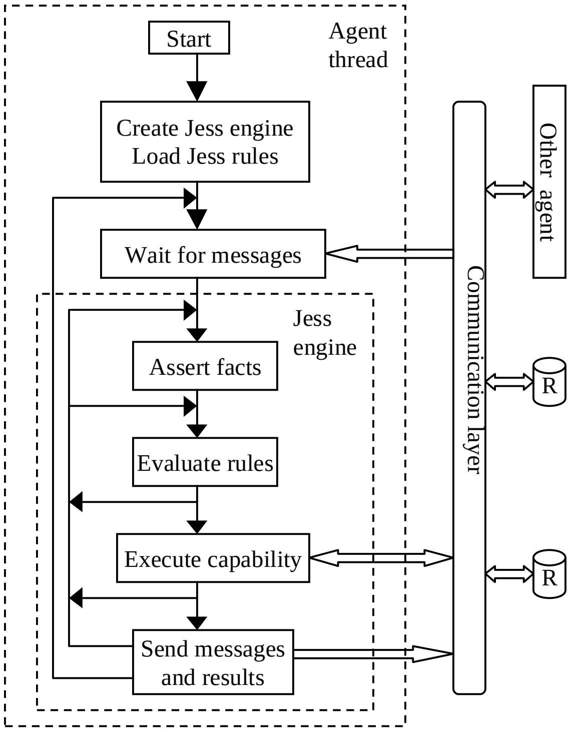

Figure 5: Execution model

whether to execute. This approach requires more messages, but is more flexible, and leaves much autonomy for the agents. Since most commitments among agents are pairwise (one agent makes commitment to the other), this point to point communication eases collaborations among agents. Therefore, we prefer the second approach.

Instead of considering each control connector in our metamodel, for brevity, we consider only a single generic version of them. We are given a connector with m inputs and n outputs. The connector fires if all inputs are received and the condition on some out-branch evaluates to `true`. This translates to a simple forward-chaining rule. When the connector requires fewer than the total number of inputs (e.g., if it is an or-join), then the rule for each of the receiving agents is a little more complex to count the number of available inputs, so it can fire when the desired number of them has arrived.

4.1 Reentrance of Connectors

After the connector condition evaluates to true and target tasks have been executed, there might be update or corrections sent from source tasks that require target tasks to execute again. We term this behavior *reentrance*.

There are two approaches for handling reentrance. One, the connector condition is always checked to ensure that the number of executions of each task reentering the connector is identical. (This test is essential for the first execution.) Two, the connector condition is not checked for executions (after the first execution). Events generated by the source tasks are sent directly to the target, which may reexecute. The second approach is more practical, but also more challenging from a theoretical standpoint, e.g., to ensure correctness.

4.2 Incorporating Commitment Patterns

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 now consider some interesting commitment patterns. P, C, refer to the producer and consumer roles, and W to the context, here the workflow itself.

Notify the consumer. This commitment pattern expressed as $\mathsf{C}(P, C, \text{COMPUTING\_DONE} \longrightarrow \mathit{notify}, W)$ comes into effect when a role finishes its execution the first time. COMPUTING_DONE is an event that signifies the "computed" state in the statechart of Figure 6. The rule is fired when COMPUTING_DONE is received. Results are sent to the given consumer and the associated commitment is created and stored.

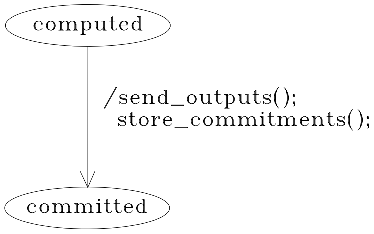

Figure 6: Notify-consumer commitment pattern

Renotify the consumer. This commitment pattern is $\mathsf{C}(P, C, \text{COMPUTING\_DONE} \wedge \mathit{!valid\_commitment} \longrightarrow \mathit{renotify}, W)$. It is enacted when P finishes its execution a second or later time. If a previous commitment to C doesn't hold, then P must tell C again and (as before) store the new commitment.

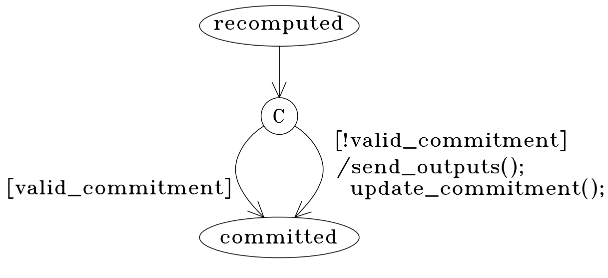

Figure 7: Renotify-consumer commitment pattern

Entertain request from another role. This commitment pattern is expressed as $\mathsf{C}(C, R, \text{START\_EVENT} \wedge$

REQUEST_EVENT $\longrightarrow$ *entertain-request*, W). 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. C is committed to P (or to W) that if P sends it an update or correction, C will reexecute the capability. Formally, $\mathsf{C}(C, R,$ START_EVENT$\wedge$ UPDATE_EVENT $\longrightarrow$ *entertain-update*, W). This pattern consists (from Figure 3) of the *committed*, *reasoning*, and *recomputed* states, and the UPDATE_EVENT transition and the succeeding transition.

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 send a complaint upstream and to demand that a property it desires in its inputs be satisfied.

5 Architecture and Implementation

Figure 8 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 3 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 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 don't forget what they did in case there are further changes. ∎

6 Discussion

Workflows are attractive, because they promise to add coherence to distributed activities in a heterogeneous, open information environment. Being able to change your mind in a controlled way is an important way to achieve progress in a dynamic, unpredictable world. 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 are employed to carry out complex activities in a coordinated manner.

6.1 Literature

There are several products and a huge body of literature on workflows. The work on rule-based systems for workflows is pertinent. A lot of it, however, deals with the lower-level aspects of workflow management. Several approaches include rules for handling exceptions [Singh & Huhns, 1994; Casati *et al.*, 1996], but they do not capture the bigger patterns of long-lived workflows.

The *language for action* metamodel involves commitments as well [Winograd & Flores, 1987], 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. The loops 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.

Several agent-based approaches exist. Klein exploits a knowledge base of generic exception detection, diagnosis, and resolution expertise [1999]. Specialized agents are dedicated to exception handling. This approach is complementary to ours, and special roles could be included in our workflows with commitments by other

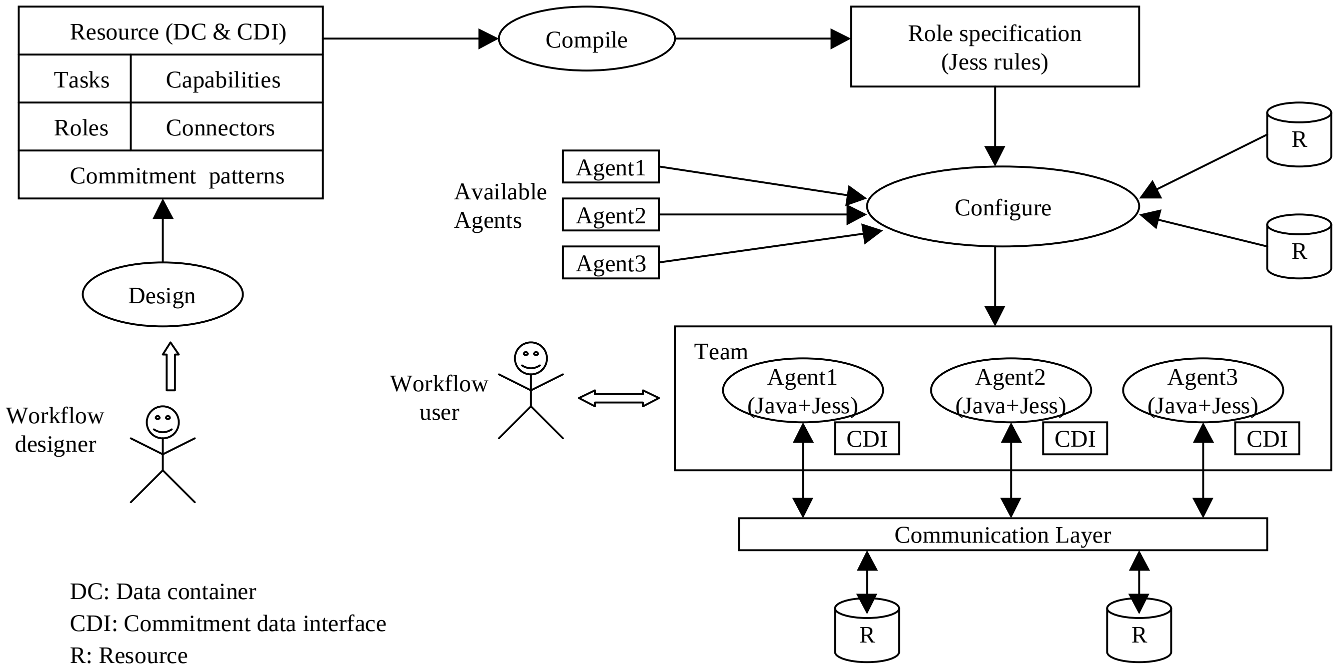

Figure 8: System architecture

roles. The advanced decision environment for decision tasks (ADEPT) project also considered workflow management [Jennings *et al.*, 1996]. This project emphasized negotiation among agents. However, the underlying notion of commitments doesn't allow contextual nesting, as in our approach.

6.2 Directions

A number of interesting technical problems are opened up by our research. One of the charms of our approach is how 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 activities in information environments and elsewhere.

References

[Bayardo *et al.*, 1997] Bayardo, R. *et al.* 1997. InfoSleuth: Semantic integration of information in open and dynamic environments. In *Proc. ACM SIGMOD Conference.*

[Casati *et al.*, 1996] Casati, F.; Grefen, P.; Pernici, B.; Pozzi, G.; and Sánchez, G. 1996. WIDE workflow model and architecture. TR 96.050, Dipartimento di Elettronica e Informazione, Politecnico di Milano.

[Fowler, 1997] Fowler, M. 1997. *UML Distilled.* Reading, MA: Addison-Wesley.

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

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

[Klein, 1999] Klein, M. 1999. Exception handling in agent systems. In *Proc. 3rd International Conference on Autonomous Agents.*

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

[Winograd & Flores, 1987] Winograd, T., and Flores, F. 1987. *Understanding Computers and Cognition.* Reading, MA: Addison-Wesley.