Warren Blanchet, Renée Elio, Eleni Stroulia. "Conversation Errors in Web Service Coordination: Run-time Detection and Repair." Proceedings of the 2005 IEEE/WIC/ACM International Conference on Web Intelligence (WI 05), September 19-22 2005, Compiègne University of Technology, France.

Conversation Errors in Web Service Coordination: Run-time Detection and Repair

Warren Blanchet, Renée Elio, and Eleni Stroulia
Department of Computing Science
University of Alberta
{blanchet, ree, stroulia}@cs.ualberta.ca

Abstract

Organizations that own web services participating in a workflow composition may evolve their components independently. Service coordination can fail when previously legal messages between independently changing, distributed components become illegal because their respective workflow models are no longer synchronized. This paper presents an intelligent-agent framework that wraps a web service in a conversation layer and a simple workflow-adaptation function. The conversation layer implements protocols and consults globally shared, declarative policy specifications to resolve interaction failures. The framework allows agents to resolves various model mismatches that cause interaction errors, including changes to required preconditions, partners, and expected message ordering. Implications of this distributed approach to web service coordination are also discussed.

1. Introduction

One of the appealing elements of web service composition is its distributed conceptualization: functional components of a system are implemented as network-accessible "services" declaratively specified (in WSDL) and declaratively composed (in BPEL4WS). However, most of the available middleware environments that support this composition assume that the coordination specification, which defines conversation controllers, is interpreted by a central middleware node. This poses some interesting issues for bottom-up Enterprise Application Integration, in which the various organizations that own the services participating in the workflow may evolve their services independently. The question then becomes "how should a composition evolve *with* its independently changing, constituent services?".

For example, suppose that Organization A adds new preconditions that must be satisfied for it to contribute its services to a workflow. To do this, it updates the original BPEL process x that specifies its contribution, thus creating x', a new version of the process which reflects the change. It also updates the BPEL process that specifies how to interact with its service, x_{client}, creating x'_{client}. However, Organization B, which relies on Organization A's services to provide its own services, still has the old BPEL process x_{client}. The result is that Organization B will invoke Organization A's service by sending an initial message without the necessary precondition being satisfied. The workflow composition itself will fail, since the message sent by B is not the message that A expects or defines as legal.

Specifically, our research examines the avoidance of workflow failures that result from out-of-sync workflow models and that are detectable through certain types of conversation errors. We view service composition and coordination as a conversation among intelligent agents, where each agent is responsible for delivering the services of a participating organization. In this context, an agent is a layer wrapping each peer organization, and is able to communicate with the other agents responsible for partner services, recognize mismatches between its own conversation model and the models of other agents (as these are revealed by conversation failures), and adapt the models as necessary to eliminate these errors.

Much of our approach is inspired by the agent communication community, which has long grappled with the matter of defining and supporting declarative specifications for the syntax and semantics of extended message exchanges (e.g., [11]). Within this community, some work argues that normative conversation behavior for agents is situated within (or defined by) a specific task model and the state of the coordinating agents in the execution of these models [3][9]. These task-situated perspectives are well suited to the convergence on a standardization language, such as BPEL [2], that supports the specifications of both composition schemas and coordination protocols.

Our approach defines an agent layer that wraps each web service, operating with a conversation layer that defines normative message exchange based on the underlying workflow model. Deviations from this normative message exchange triggers a conversation error, which is regarded by both agents as a symptom of mismatching workflow models. The agents consult policies to resolve whose model is to be used as the correct one, and then restart their interaction.

The rest of this paper is organized as follows. Sections 2 and 3 discuss how workflow and conversation models can define conversation errors as symptoms of mismatched workflow models. In section 4, we discuss

the role of declarative policies, indexed to error and workflow types, for identifying which agent has the correct workflow model. Sections 5 and 6, respectively, describe our implemented architecture for workflow failure avoidance and two illustrative case studies. Section 7 discusses related research. We conclude with a discussion of open issues and directions.

2. Workflows, Conversations, Models, and Scripts

A *workflow* is a process that accomplishes some (business) objective and that requires more than a single participant. Therefore, it involves routing information among participants so that the objective can be accomplished. A *workflow model* is a conceptualization of a workflow, typically represented as a state space in which the edges represent actions taken by the workflow's participants. These actions include both private (agent internal) computations and message exchanges between participants. We assume that a workflow model includes partner definitions and roles. (This is important for the conversation error types we detect). We use the term *workflow script* to mean the states and state transitions (steps) of a particular participant *p* in a workflow *w*, from *p*'s perspective, including message exchanges with other participants. Each workflow script is stored as a BPEL specification extended by partner bindings and roles. A workflow model that involves *n* participants will be realized as *n* workflow scripts, one for each participant.

We use the term *conversation* to denote the exchange of messages between a sender and receiver, initiated by the sender to bring about a particular state change. A *conversation script* is just that portion of a workflow script *w* that specifies message exchanges between the process modeled in *w* and some particular partner process. Thus, if an agent has multiple partners, there will be a number of corresponding conversation scripts that, when combined with the agent's internal work activities, constitute the workflow script of the agent; the workflow scripts of all collaborating agents together correspond to the workflow model. To create conversation scripts, we map the web-service primitive message interaction types (one-way, request response) from the workflow script to a small set of protocols that use higher-order message primitives (see Table 1).

The structure of these higher-order messages is (*:message-primitive :receiver r :sender s :conversationID cid :content c).* We use three message primitives, *inform*, *request*, and *not-understood*, adapted from [10]. *Inform* is used by a sender agent to communicate information to a receiver agent, when the sender's state indicates that the receiver expects or requires that information. *Request* is sent by the sender to request that the receiver perform some action. *Not-understood* is our interaction error primitive. The receiver and sender fields hold unique agent ID's, specified as URI's. The conversation id is dynamically set by the initiator of the conversation script, and designates a unique conversation thread. The structure of the content depends on the message primitive used. For *inform*, it is a tuple consisting of an *operation ID* and the *message content* proper. The operation ID identifies a WSDL-level operation. It subsumes the service, port, and operation identifiers present in WSDL. The message content for *not-understood* consists of a 'reason' for regarding message *m* as an error in the context of conversation *cid* with sender *s*. We discuss the taxonomy of reasons and the *not-understood* message in more detail later.

BPEL Activity	***Message Protocol***
Asynchronous Invoke	Send *inform*
Asynchronous Receive	Receive *inform*
Synchronous Invoke	Send *inform* Send *request* Receive *inform*
Synchronous Receive	Receive *inform* Receive *request*
Synchronous Reply	Send *inform*

Table 1: Mapping BPEL Activity to Conversation Protocols

We interpret any message sent between web services as information of one kind or another, and thus for these we use *inform*. Consider the case of a synchronous BPEL *invoke* activity (the type that invokes a WSDL *request-response* operation). The invoker initiates the sequence by sending a message to the receiving process. Due to the nature of the WSDL operation, the receiving process must reply with a message sent to the invoker, who is waiting for the reply. Following Table 1, this exchange is translated to the following protocol: the requester sends an *inform* message (with the WSDL input message as content) immediately followed by a *request* message (requesting the WSDL output message). The invoker of the operation is actively soliciting a response, as is clear from the semantics of the WSDL request-response operation, and thus we use a *request* for action (the request for the action of sending back a response). The receiver sends an *inform* message (with this message as content) as a reply.

Our reason for moving to this higher level of message protocol is to enable a more flexible layer that recognizes conversation errors as symptomatic of mismatched workflow models.

3. Conversation Failures and Errors

A conversation failure is the failure of a conversation to result in the state change *c*, for which it was selected. Conversation failures arise from conversation errors: deviations from the normative behavior specified by a given conversation script. A conversation error (if unresolved) typically results in conversation failure, which in turn leads to a workflow failure. A conversation

error is defined from a particular participant's viewpoint—it is a deviation from the normative message exchange as defined by *its* conversation script, which is derived from *its* workflow script. When this occurs, the participant constructs and sends a *not-understood* message.

4. Policies for Conversation Error Recovery

There are four components relevant to our approach to conversation error recovery: (a) the error symptom (a conversation error observable through message exchange); (b) the cause for the error (model mismatch); (c) a way to fix the underlying models, so that the conversation error does not reoccur; (d) a selection policy: a method of determining which underlying model to adjust and how. There might be more than one cause for a given conversation error, but our current approach does not require these be diagnosed and treated differently. If an agent *A* receives a message *m,* it would generate a *not-understood* in response to the following error symptoms:

1. *A* cannot interpret *m*'s content. *Causes*: The workflow scripts of *A* and *m*'s sender differ on the schema specifying constraints for *m*'s content or on the form of the information provided in *m*, or the inner message primitive is not known to *A*.
2. *A* does not expect *m* at this point. Either *m* is routable to an ongoing conversation *c*, but is not the next legal message given *c*'s state, or belongs to no open conversation and is not a legal conversation-starting message. *Causes*: The message exchange involving *m* was deleted from the receiver's conversation model, a new message exchange was added to the receiver's conversation model before *m*, the message exchange involving *m* was reordered in the receiver's conversation model, or a message exchange has been substituted for *m*'s in the receiver's conversation model.
3. *m* is sent by a wrong party: *m*'s message type and content are expected and interpretable in an on-going conversation *c*, but the sender is not the partner expected by *A*. *Causes*: the roles of the partners have evolved; partners have delegated their responsibilities to other agents.

A *not-understood* message, with a reason as its content, constitutes a kind of run-time error about the nature of a model mismatch. The reason identifies one of these error categories. To recover from this potential workflow failure, then either the receiver's workflow model must be modified or the sender's workflow model must be modified, so that either *m* is not regenerated by the sender, or *m* is regarded as legal by the receiver.

A *policy* is a means by which one of two different workflow models for accomplishing some objective will be regarded as correct, and adopted by the agent with the incorrect model, so that the interaction can continue. Our policies are either general defaults that apply to multiple workflows, or are indexed to particular error types within particular workflows. A general default policy, for example, might be a kind of virtual 'organization chart', in which hierarchical position defines authority relationships among agents. The need for workflow and error-type specific policies for designating authority will be illustrated in case studies, discussed below.

5. WRABBIT System Implementation

We have implemented this approach to interaction-failure recovery in a system called "Workflow Reconfiguration with Agent- and BPEL-Based Intercommunication Technology", or WRABBIT. Figure 1 illustrates the software architecture of each WRABBIT agent. Our focus here is on the agent layer and the way it recognizes conversation errors, as well as the way it supports dynamic adjustment of workflow scripts to avoid such errors so that the workflow can continue.

There are three key elements to how the agent layer attempts to re-synchronize workflow models: an algorithm for composing workflow scripts into executable meta-scripts, the recognition and classification of a conversation script error (generating a *not-understood)*, and the use of declarative correction policies associated with the workflow and error type (resolving a *not-understood).*

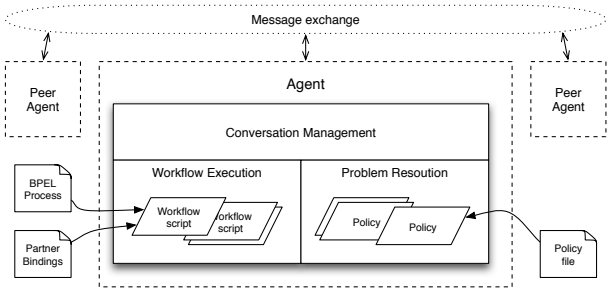

Figure 1: The WRABBIT Architecture

WRABBIT agents are designed to achieve objectives of two general types: objectives to bring about a state *s* and objectives to execute action *a.* Agents create an objective of the first type to obtain the value of a particular information type (currently, specified as a WSDL message type) and this leads to creating a workflow script. As soon as it has created a workflow script, the agent creates the objective to execute it – an objective of the second type. Setting an objective means creating an instance of one of these objective types and allocating resources until it is achieved or determined to be impossible.

Essentially, the agent layer operates as a loop, setting objectives, selecting one objective, and then advancing work on that objective. It halts when all objectives are satisfied. The three most important kinds of objectives are those concerned with processing received messages,

generating workflow scripts, and executing workflow scripts. Message handling takes precedence: the message queue is examined first for unprocessed messages and a *message-dispatch objective* is set to determine whether *m* is legal and expected for some currently open conversation with *m*'s sender, or whether *m* is a legal start to a new conversation with *m*'s sender. If neither is the case, then the agent generates a *not-understood* message as a reply to *m*, with a reason that corresponds to one of the error categories described above.

Finally, *m* itself could be *not-understood*, sent as a reply to some previous message the agent sent. When an agent receives a *not-understood,* and also when it generates one, it sets a *not-understood-resolution* objective, aimed at resolving the not-understood message associated with some particular conversation and some particular partner.

Since the workflow composition algorithm is used in resolving *not-understood* messages, we describe it first.

5.1 The workflow composition algorithm

Our workflow composition algorithm works iteratively to construct a meta-workflow script that matches a set of conditions. Complex semantics-based matching for web-services has been explored by Aggarwal et al. [1], and is not the focus of our work. Therefore, the current implementation features a simple composition algorithm, which performs elementary matching using information types (currently specified as WSDL message types). In addition to the information types required, our algorithm is provided with the set of workflow scripts known to the agent. Some workflow scripts are primitive actions (or in BPEL parlance, are executable), in that they can be directly executed without further variable binding. Other scripts are not primitive (or in BPEL parlance, are abstract or protocols), in that they require values for information types for their execution. The workflow composition algorithm constructs an executable meta-workflow script that consists of any number of these abstract or primitive scripts, in a particular order. The ordering ensures that, when the meta-script is executed, all the information requirements and control dependencies of the constituent scripts are satisfied.

The algorithm takes a standard problem decomposition approach, searching its collection of workflow scripts for one that provides the desired value. The executable scripts are preferred and hence are searched first. If a suitable executable script is not found, the abstract scripts are then searched. Since values of other information types are required to construct an executable meta-script that contains an abstract script *s,* the algorithm is re-invoked with script *s*'s information types as input. The algorithm halts when it has found or constructed executable scripts that satisfy each information type identified, or after its search has failed. The algorithm uses a script-ranking method to ensure that only one copy of a given script (that resolves some dependency) is inserted at the right place in the execution order.

5.2 Resolving not-understoods

Having created a workflow meta-script, the agent layer extracts one or more conversation scripts from it, using the mapping from BPEL message interaction types to higher-order protocols, described earlier. Agents consult these conversation scripts to determine if a message is legal, within the current context of a workflow execution. If it is not, a *not-understood* is generated with a reason that corresponds to one of the error categories described earlier.

To resolve a *not-understood*, both agents consult a commonly-held declaratively specified correction policy, indexed by workflow and conversation error types within workflows. As noted earlier, a policy is some means by which one or the other of the two differing workflow models will be used as the current, correct one. This reduces to specifying a source for new workflow files. (There could be more complex policies that require an even longer message exchange, but we ignore those here). From either agent's perspective, if the source of the correct workflow is itself, then it does nothing, as all agents have an objective to send workflow scripts when asked; if the source is the partner, then it requests new scripts from that partner; if the source is a third party, then both agents send requests to that third party for new workflow scripts. Note that *not-understood* and its subsequent resolution constitutes a new conversation, and the protocol for that conversation is implicit in the policy (this is why both the generation and the receipt of a *not-understood* triggers an objective to resolve it: both the sender and receiver are prepared to exchange further messages aimed at its resolution). If new scripts have been received, the agent layer again uses the workflow composition algorithm to generate a new meta-script, resolving any new dependencies. The agent that initiated the conversation that led to the *not-understood* message will re-initiate the conversation, and the same conversation error will not occur.

6. Case Studies

We have run various case studies that exercise our approach and identify issues for further investigation. We present three cases here, inspired from intuitive scenarios of workflow reconfiguration as it might happen in academic departments. They involve four agents: a *DepartmentAgent* performs the functions of a member of the department's administration, a *PayrollAgent* accomplishes tasks that are handled by a university's payroll group, as well as an *InstructorAgent* and a *TeachingAssistantAgent* that carry out the activities of these faculty members. Our concerns, generally speaking, revolve around the failures that occur when a service

provider's workflow is redefined, but the service's client continues to operate under the old model, or vice versa.

6.1 Missing Preconditions

For the first case study, the *InstructorAgent*'s objective is to obtain a value of the student transcript type. It thus selects an executable workflow script that produces a student transcript message (composition is not necessary at this point), and sets an objective to execute this script. This script contains a conversation script with the *DepartmentAgent* and the *InstructorAgent*, and the messages exchanged are depicted in Figure 2.

In this scenario, however, the *DepartmentAgent's* workflow for releasing transcripts has been redefined to require an authorization token (which it provides with a separate service) prior to releasing transcripts (e.g., as per some new university regulations). The initial message from the *InstructorAgent*, which requests the transcripts before providing an authorization token, causes a conversation failure (Figure 2(a)).

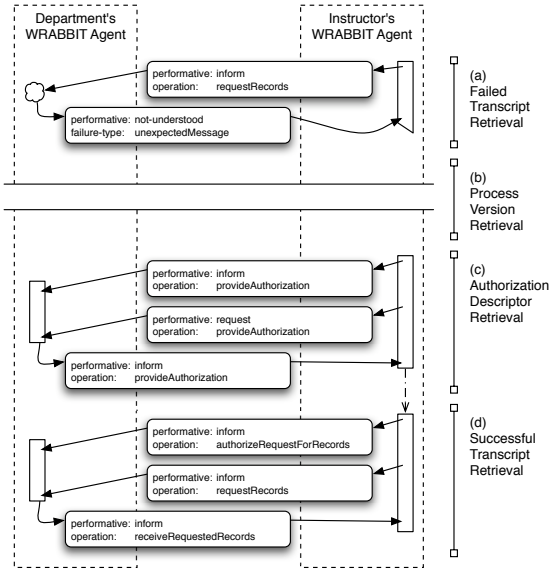

Figure 2: Scenario 1 Message Exchange

The error symptom here is that *InstructorAgent'*s message is unexpected, as it no longer initiates a legal conversation, from the *DepartmentAgent'*s viewpoint. The *DepartmentAgent* sends a *not-understood* message to the instructor's agent identifying this failure symptom, and both processes are terminated. The *InstructorAgent*, using the content of the *not-understood* message, determines that its "Student Transcript Retrieval" workflow script does not match the "Student Transcript Disbursement" workflow script of the department's agent. The policy dictates that the *DepartmentAgent* is the authority for this type of error (unexpected message) for this particular workflow ("Student Transcript Exchange"). The *InstructorAgent* initiates a message exchange with the *DepartmentAgent* to obtain an updated set of the files that define its workflow script (Figure 2(b)). (Note that it is a requirement for all agents who may act as an authority to provide scripts when requested. Thus, they are configured to execute conversations with other agents to provide any scripts for which they serve as authority.) Once this is done, the *InstructorAgent* uses the workflow composition algorithm to construct a new workflow script. When the algorithm selects the (new) "Student Transcript Retrieval" script, it identifies the new precondition of obtaining the authorization descriptor. In this scenario, the *InstructorAgent* was configured with a workflow script that retrieves this descriptor from the *DepartmentAgent*, and the workflow composition algorithm inserts it into the meta-script such that it will execute prior to the "Student Transcript Retrieval" script. Because the respective conversation scripts for the *DepartmentAgent* and the *InstructorAgent* now match, the conversation completes successfully (Figure 2(d)).

6.2 Changes to Partners

The next case study has a similar theme: the *DepartmentAgent* provides a student transcript provision service to the *InstructorAgent*. The difference here is that the *InstructorAgent* modifies the service to additionally allow access to the *TeachingAssistantAgent*. After this service modification, the *InstructorAgent'*s BPEL specifications and support files are copied to the *TeachingAssistantAgent*, enabling it to create workflows to obtain student transcripts. Thus, the *TeachingAssistantAgent* becomes configured to operate as a legitimate partner in the transcript provision service.

The case study begins with the *TeachingAssistantAgent* intending to obtain student transcripts. It selects an executable workflow script that produces a student transcript message (further composition is not necessary in this scenario), and sets an objective to execute this script. This script is the one containing a conversation script with the *DepartmentAgent* to obtain a transcript. However, the *DepartmentAgent* expects the requester of transcripts to be one of a set of agents, and according to its specifications, only the *InstructorAgent* is a member of that set. The *InstructorAgent*'s addition of the *TeachingAssistantAgent* to this list has created an updated workflow model that is not synchronized with the *DepartmentAgent'*s model.

The conversation error occurs when the *TeachingAssistantAgent* sends a request to the *DepartmentAgent* for the transcripts. The *DepartmentAgent* consults its workflow script, and finds that *TeachingAssistantAgent* is not on the list of allowed partners for this operation. It replies with a *not-understood* message identifying the reason as "message from wrong party".

The resolution of this mismatch requires all three agents, even though the *InstructorAgent* was not part of the original conversation. In this scenario, the default policy specifies that the *DepartmentAgent* is the authority on the student transcript workflow. However, an error-specific policy associated with "message from wrong party" within this workflow indicates that the *InstructorAgent* is the authority on the allowed agents for the workflow. This allows the *InstructorAgent* to delegate roles and responsibilities to other agents. For our case study, this corresponds intuitively to a course instructor delegating certain responsibilities to a student acting as a course TA, who would not otherwise be able to request transcripts.

Both the *DepartmentAgent* and the *TeachingAssistantAgent* retrieve the policy indexed to this error type within this workflow. Each obtains the updated scripts from the *InstructorAgent.* (As noted earlier, it is a requirement for all agents who may act as an authority to provide scripts when requested.) These updated scripts include updated partner files that incorporate the modification. The *TeachingAssistantAgent* reinitiates the request for student transcripts (the *DepartmentAgent* queues the request, if its updates have not arrived) and its workflow executes successfully.

6.3 Message reordering

In this last case study, a service provider reorders two message exchanges in the conversation with its service consumer. The *PayrollAgent* provides a service to mail paychecks to employees of the university. This service requires the address of the employee, the employee's salary information, and a note to print on the paycheck (e.g. a seasonal greeting, or a reminder). In return, the estimated date of check arrival is returned. In the original specification of the service, the address, salary information, and note were to be provided in that order. Suppose that that *PayrollAgent* modifies its service, so that it requires the salary information first and the address information second.

The *DepartmentAgent*, unaware of this service modification, sets an objective to send a paycheck to an employee, and initiates this interaction with the *PayrollAgent* by sending the address information first.

The exchange is diagrammed in Figure 3. The *DepartmentAgent*'s initial message is expected by the *PayrollAgent* as the second message, and so the *PayrollAgent* responds to it with a *not-understood* message (Figure 3(a)). The failure type here is "unexpected message", because the message was not expected at the time it was received. However, before the *DepartmentAgent* receives this *not-understood*, it continues sending out messages according to its (out of sync) conversation script. Upon receipt of the *not-understood* from the *PayrollAgent,* the *DepartmentAgent* closes its conversation thread and begins the error resolution process. Meanwhile, the *PayrollAgent* recognizes the *DepartmentAgent*'s second message as *starting* a paycheck mailing conversation (as per its updated service specification). Of course, the *DepartmentAgent*'s third message in this stream is not the correct second message, from the *PayrollAgent*'s viewpoint. Thus, the third and fourth messages also receive replies of *not-understood* (see Figure 3).

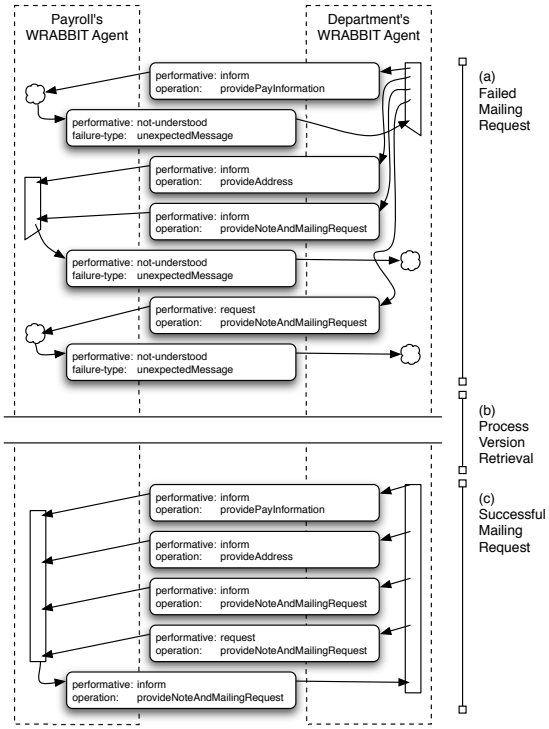

Figure 3: Scenario 3 Message Exchange

To resolve the conversation failure, the agents consult the shared policy. For this type of error (unexpected message) for this particular workflow ("Student Transcript Exchange"), the *PayrollAgent* is the authority. Thus, the *DepartmentAgent* obtains the necessary documents from the *PayrollAgent* to construct an up-to-date workflow script (Figure 3(b)). Once the *DepartmentAgent* has the updated workflow script, it sets the objective of mailing a paycheck again, and in this instance will succeed, because the workflow scripts are compatible (Figure 3(c)).

7. Related Work

This work is predicated on a distributed, bottom-up approach to web service composition and coordination, in contrast to centralized execution models. The community is still debating the issue of centralized vs. distributed coordination of web-service compositions [16] Centralized execution models constrain the broad vision of the web-service composition paradigm, sometimes resulting in fragile systems (for example, when the central

node becomes unavailable, the composition breaks down) and imply heavy network traffic and poor performance. Although most mature middleware support for BPEL process execution assumes a centralized view, distributed environments are receiving increasing attention. The Symphony project [14] aims to address the brittleness and inefficiency shortcomings of centralized workflow-execution engines: it uses an algorithm for analyzing a BPEL workflow specification for data and control dependences among the constituent services and partitioning it into a set of simpler BPEL specifications, each one corresponding to an individual service. These components are then distributed to different locations and, when deployed, cooperatively deliver the same semantics as the original workflow. Symphony does not provide any support for failures arising from workflow mismatches since it assumes that the distributed processes will be derived from a single complete BPEL process.

The WARP environment [5] uses agents' reflection and tuple-space communication to coordinate a workflow of component-based services. The COACHES approach [4] investigates how to organize agents in groups in order to enable their better collaboration during workflow execution. Additionally, the COACHES agents can invoke web services, providing an alternative to the web services standards for workflow. However, this work does not address the problem of misaligned workflows that we have considered.

Buhler and Vidal have also used agents for the execution of workflows specified in BPEL [6]. Once again, however, the current focus of that work seems to be distributing the execution of the business process among agents. While they share our opinion that an agent system's adaptability is an asset, their implementation efforts have yet to capitalize on this ability [7].

A *web-services choreography* specification [17] is under development, with the goal of providing a common abstract language for describing legal and expected communication. Our work adopts this premise and examines a specific consequence: how to support the dynamic recovery from workflow coordination failures due to mismatching workflow models.

In the area of general distributed workflow execution and management — outside web services specifications — there has been considerable work on workflow change management. For example, [12] presents a distributed workflow management system, in which agents both execute the workflow and manage state information. The agents in this system are of different types, where we adopt a peer-to-peer approach, and their focus is on recovering from application-level failures that result in the inability to deliver a service, not from the misalignment of independently evolving workflows.

There is similarly a substantial body of work on intelligent-agent conversation and collaboration, involving standardization proposals for agent communication languages, protocols, and conversation policies [10][11][13]. All these are different levels of granularity in defining normative interaction behavior. The use of dynamically initiated task-independent conversation protocols to handle error conditions, whenever necessary, has been explored in (e.g., [15]). Other research [9] leverages a jointly held task model as a means of defining normative interaction and hence interaction errors, a concept this work has extended by adding error correction policies. COOL [3] is a programming language that defines agents first in terms of their conversational interfaces. Both these approaches view a conversation as just another action an agent selects to accomplish some objective, and the legality or plausibility of such an action is situated in the current context of the coordination effort (e.g., what other information has been exchanged so far, what each partner expects the other partner to require or expect right now, etc.). Finally, the view of web-service composition and coordination as a distributed agent problem places it squarely in the realm of multi-agent coordination and cooperation research, which has a number of mature methodologies to offer, particularly in the realm of coordinated scheduling and commitments to perform services [8].

8. Discussion and Future Directions

A robust web-service composition infrastructure will have to entail extended message exchanges (conversations) with more complex content, aimed at dynamically recovering from coordination failures. This work presents a design and implementation of a framework for recognizing and resolving workflow coordination failures, based on a taxonomy of conversation error types. The framework wraps web services in an agent layer that uses simple conversation protocols (mapped from BPEL message exchange types) and the error taxonomy to detect unexpected messages and provide 'reasons' for the conversation error. This work assumes that such failures occur through mismatching workflow models, and that some agent is the source of model information that, if adopted, will eliminate the conversation error. Declarative policies, either general or indexed to particular workflows and error types, are used to identify which agent serves as the authority and source for the correct model. The implemented framework has been tested on a number of case studies.

A strongly distributed perspective on web service composition and coordination entails, we think, the possibility that some components of the composition will evolve independently, leading to interaction errors. The kind of approach presented here aims to recover dynamically from such interaction errors and do so by isolating the recovery effort. The error is detected and resolved between two specific partners with the out-of-sync models and other partners will not be affected. This

will aid distributed management, where the owners of smaller workflows can evolve these fluidly, without concerning themselves with the higher-level workflows in which theirs are components. Our on-going work concerns assessing the robustness of this approach when applied to a larger number of coordination errors, spanning a larger number of agents, which must consult more complex policies for determining which agent is the authority for correcting some particular workflow model.

A distributed agent approach, while solving some of the problems associated with centralized approaches, brings its own challenges. For one, it requires some homogeneity of the distributed agents, who are attempting to act as peers. As the agent communication community has discovered, this homogeneity goes beyond standardized message types and message-exchange protocols, for there is no guarantee that the intended semantics associated with messages and protocols are actually implemented in each of distributed agents. Within the context of our own work, we acknowledge that at least *one* thing must be centralized, namely the declarative specification that associates particular policies with particular conversation errors for (possibly) particular workflow models. If this is achieved, then there is the chance for some automatic repair of interaction errors at the infrastructure level, allowing applications and their designers to focus their efforts on application-specific problems. The ability to dynamically recover from mis-matched coordination models becomes more complex if agents are not guaranteed (as we have assumed so far) to be able to resolve any new dependencies that emerge when working with update workflow models. This will cause the workflow composition algorithm to fail, and the worst case is that an overall coordination failure has just been delayed, not avoided. However, this is the sort of issue that will arise repeatedly with any system predicated on distributed intelligence: centralizing declarative policies or joint coordination models written in a standardized language still require that the distributed agents be implemented to interpret them and behave consistently.

9. References

[1] R. Aggarwal, K. Verma, J. Miller, and W. Milnor. "Constraint Driven Web Service Composition in METEOR-S." In *Proceedings of IEEE International Conference on Services Computing*, 2004.

[2] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. "Business Process Execution Language for Web Services, Version 1.1." Specification. BEA Systems, IBM Corp., Microsoft Corp., SAP AG, Siebel Systems (2003).

[3] M. Barbuceanu and M. S. Fox. COOL: A language for describing coordination in multi agent systems. In *Proceedings of the First International Conference on Multi-Agent Systems*, pp. 17-24, San Francisco, California, 1995.

[4] M. B. Blake, "Forming Agents for Business Process Orchestration." In *Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 7*. 2004. p. 70210a.

[5] M. B. Blake, "WARP: An Agent-Based Cross-Organizational Workflow Architecture in Support of Web Services." In *Proceedings of the 2000 International Conference on Artificial Intelligence (IC'AI2000)* Las Vegas, NV: CSREA Press Science.

[6] Paul Buhler and José M. Vidal. "Towards Adaptive Workflow Enactment Using Multiagent Systems." *Information Technology and Management Journal*, 6(1):61--87, 2005.

[7] Paul Buhler and José M. Vidal. "Enacting BPEL4WS Specified Workflows with Multiagent Systems." In *Proceedings of the Workshop on Web Services and Agent-Based Engineering*, 2004.

[8] Decker, K. and V. Lesser. 1995. Designing a family of coordination algorithms. In *Proceedings of the First International Conference on Multi-agent Systems*, pp. 73-80, San Francisco, California.

[9] R. Elio and A. Petrinjak, "Normative communication models for agent error messages", *Autonomous Agents and Multi-Agent Systems*, (in press).

[10] FIPA Agent Communicative Act Library Specification, www.fipa.org.

[11] M. Greaves, H. Holmbeck, and J. Bradshaw, "What is a conversation policy?" In *Issues in Agent Communication (LNAI 1916). Edited by* F. Dignum and M. Greaves. Springer-Verlag, Berlin. 2000. pp. 118-131.

[12] M. Kamath and K. Ramamritham, "Pragmatic Issues in Coordinated Execution and Failure Handling of Workflows in Distributed Workflow Control Architectures." Univ. of Mass. Computer Science Tech Rep 98-28, Aug, 1998.

[13] Y. Labrou and T. Finin, "A proposal for a new KQML specification," Technical Report #CS-97-03, Computer Science and Electrical Engineering Department, University of Maryland, Baltimore, Maryland. 1997.

[14] Mangala G. Nanda and Neeran M. Karnik. Synchronization Analysis for Decentralizing Composite Web Services. *International Journal of Cooperative Information Systems*, vol. 13, no.1, March 2004, pp 91--119.

[15] M. H. Nodine and A. Unruh, "Constructing robust conversation policies in dynamic agent communities". In *Issues in Agent Communication (LNAI 1916). Edited by* F. Dignum and M. Greaves. Springer-Verlag, Berlin. 2000. pp. 206-219.

[16] C. Peltz, "Web services orchestration." Hewlett-Packard Technical Whitepaper, Jan 2003. http://devresource.hp.com/drc/technical_white_papers/WSOrch/WSOrchestration.pdf.

[17] Web Services Choreography Description Language Version 1.0, W3C Working Draft December 2004, http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217/