Blanchet, W., Stroulia, E., & Elio, R. (2005) Supporting adaptive web-service orchestration with an agent conversation framework. In *Proceedings of the 2005 IEEE International Conference on Web Services*, 541-549. July 11-15, Orlando, USA.

Supporting Adaptive Web-Service Orchestration with an Agent Conversation Framework

Warren Blanchet
University of Alberta
blanchet@cs.ualberta.ca

Eleni Stroulia
University of Alberta
stroulia@cs.ualberta.ca

Renée Elio
University of Alberta
ree@cs.ualberta.ca

Abstract

Service-oriented architecture is emerging as a compelling paradigm for developing web-based software applications. In this style, the functional components of the system are implemented in various programming languages as network-accessible "services" declaratively specified (in WSDL) and declaratively composed in workflows (using BPEL4WS). Despite this fundamentally distributed conceptualization of service composition, most current middleware assumes that the specification of the service composition is interpreted at run time by a central middleware node. This implies inflexible composition evolution: all parties must be updated concurrently to avoid interaction failures. This paper introduces an intelligent-agent framework that wraps web services in a conversation layer and is capable of a simple workflow-adaptation function. The conversation layer implements protocols and consults globally shared, declarative policy specifications to resolve conversation failures. Two case studies illustrate this approach.

1. Introduction

Service-oriented architecture is the emerging paradigm for developing web-based software applications. This new style dictates that the functional components of the system are implemented as network-accessible "services" declaratively specified (in WSDL) and declaratively composed (in BPEL4WS).

In principle, this paradigm applies equally well to designing and developing new applications (top down "green-field" application development) and to integrating modules of existing applications (bottom up Enterprise Application Integration). In practice, several available middleware environments are capable of using web services, interpreting the specifications of their interface and composition, and choreographing their execution, though the actual implementations of the services may exist anywhere on the web.

Despite this fundamentally distributed conceptualization of web-service composition, most current middleware assumes that although the services are distributed, the specification of their composition at run time is interpreted by a central middleware node. Such centralized execution models constrain the original vision of the paradigm. More importantly, they result in fragile systems (for example, when the central node becomes unavailable, the composition breaks down) and imply heavy network traffic and poor performance. These last two shortcomings have already been the subject of study: Chafle et al. [6] proposed a technique for partitioning a composite web service specified as a single BPEL process into an equivalent set of decentralized processes. Their partitioning algorithm aims at minimizing the communication costs and maximizing the throughput.

There is another, even more insidious problem with the centralized-orchestration assumption. It implies that any changes to the composition should be made to the single copy of the declarative composition specification maintained by the orchestrating node. This is fundamentally unrealistic for bottom-up Enterprise Application Integration, where the various organizations that own the services and sub-workflows participating in the composition are likely to evolve these services independently, changing the type or sequence of messages sent and received. The effects of these changes would likely be limited to some subset of the available services. However, any change would need to be implemented as a modification of the centrally managed specification. As a result, for a sufficiently complex workflow, the specification would act as a bottleneck for the implementation of these otherwise localized changes. Although there are ongoing standards-development efforts to specify how to manage service versioning, these do not address how compositions established with a set of services might evolve with their components. Some alternative is necessary, as uncontrolled independent evolution will lead to interaction failures if a workflow's participants have different expectations regarding the sequence and type of messages to be exchanged.

Our work focuses on the distributed detection of conversation failures that arise from independently changed workflow models and on specifying policies and procedures for eliminating the cause of the failure. When an agent receives a message that is illegal by its own communication model, an exception-handling conversation is triggered. This event establishes that the two agents have mismatched workflow models, and the goal of the exception-handling conversation is to re-

synchronize these models. This policy allows automatic repair of workflow model mismatch at the infrastructure level, allowing applications and their designers to focus their efforts on application-specific problems.

Our approach views service orchestration as a conversation among intelligent agents, each one responsible for delivering the services of a participating organization. In this context, an agent is essentially a layer wrapping each peer organization. This agent is able to communicate with the other agents responsible for partner services, recognize mismatches between its own workflow model and the models of other agents revealed by conversation failures, and adapt the models as necessary to eliminate these errors. In this paper, we explore a small class of conversation errors that arise due to mismatching workflow models between two or more interacting services, and illustrate our approach for handling such errors and allowing continued operation.

The rest of the paper is organized as follows. Section 2 argues that workflows based on web-service composition can be viewed as conversation models among intelligent agents. Section 3 describes the software architecture of our framework, WRABBIT, for web-service orchestration. Section 4 illustrates the capabilities of the WRABBIT agents with two examples. Section 5 places this work in the context of related research and Section 6 concludes with an overview of the lessons we have learned to date and our plans for future work.

2. Workflows as Conversation Models

The purpose of a workflow is to deliver a work product. The reason that messages are exchanged between entities during the execution of workflows is that the entities have different abilities, such that no single entity can complete the workflow alone. Therefore, the reason that any entity sends a message *m* to another entity is that the workflow requires that some step *s* be performed, and the receiver of *m* will perform *s* as a result. In this manner, the conversation between the entities is tied to the advancement of the workflow. In the rest of this section, we consider how this basic concept applies to web-service composition.

2.1. Agent Conversation Models

Agents are aware of the purpose of the workflow they are executing, and thus their conversations are directly tied to the workflow's advancement. When discussing agents, we use the term "conversation" to denote an agent interaction that is initiated by a sending agent to satisfy some purpose; these messages follow the syntax and semantics of some implementation-independent agent communication language (ACL). Abstractly, a message is an action that is attempted by the sender on the internal state of the receiver in order to satisfy this purpose. A normative conversation model specifies the conventions shared by agents when exchanging messages. A simple example of one element of such a model is a *protocol*, which captures the message-sequencing constraints between a sender and a receiver. Communication models are actually portions of a workflow model (which also includes the non-communicative work performed by the agents), but are interesting enough to study alone.

The agent-communication community has long grappled with what should or could go into such normative conversation models, under the general theme of conversation policies [9]. Such policies were envisioned to be public, declaratively specified constraints on the "nature and exchange of semantically coherent … messages," existing separately from the agent implementation, but presumably interpretable by all interacting agents. In [9], the authors note that any particular conversation will be governed by several policies (e.g., policies for interpreting a timeout or a missing acknowledgement; termination policies; exception-handling policies; and specific goal-coordination policies, such as requesting or providing services within particular time constraints). This kind of policy specification would outline all elements of a normative conversation model. Deviation from the model is a reason to throw an exception, i.e., indicate that a particular message is illegal, "unexpected," or "not understandable" at some particular point in the conversation. Robust and flexible methods are required to address these conversation failures, so that designers do not have to anticipate them and accommodate for their presence in the set of pre-defined, fixed protocols. Some frameworks take a building-blocks approach, in which agents glue together small protocol units dynamically during run time (e.g., [14]); others reply on common representations of the joint task to define an error space [7]. Generally speaking, the aim is to delegate some portion of the error-recovery effort from the agent designer to the agent itself, in an application independent manner.

2.2. Web-Service Workflows

In the current standards used for web service composition, the perspective of a workflow as a conversation is implicit, albeit weak. For example, at the BPEL process-specification level, the workflow partners are not explicitly coordinating or cooperating: each partner process invokes its constituent services in the appropriate order and sends (receives) the messages that its partners expect (produce). The organizational workflow advances without the participants being aware of it.

Another weak point is that BPEL process specifications capture the workflow model from the perspective of one partner only: if there are *n* interacting partners, there will be *n* BPEL specifications, each one modeling only the conversations that involve that partner and the message exchanges from this partner's viewpoint. Collectively, however, these partial workflow models do contain the elements that one would find in a global workflow model. In this work, we refer to the portion of a workflow model that specifies one partner's contribution as a *workflow script*. The corresponding fraction of the conversation model is called a *conversation script*.

The shift in interest from web-service composition to orchestration brings with it an explicit element of coordination, particularly when the services are distributed. The belief is that a robust, dynamic composition of distributed services will entail extended message exchanges with more complex content to support that coordination [15]. We note that a *web-services choreography* specification [16] is under development, with the goal of providing a common abstract language for describing legal and expected communication. The draft specification provides an example list of conversation failures, ranging from syntactic errors in message construction to "application failures" (e.g., failure to complete an order because the ordered goods were out of stock). In this regard, the web-services choreography specification provides, essentially, the high-level language for defining a conversation policy of the sort envisioned by Greaves et al. [9] for distributed agents whose domain-level task requires a robust coordination and composition of web services. Given that such a language is being developed, we focus our work on the mismatches of the global communication model that includes all partners and that is reflected in the per-partner BPEL specifications.

We use the term *conversation failure* for the case where an agent receives an unexpected message type, message content, or message parameters from another agent, as part of a conversation. How could this occur? We assume that the owner of a web service is the authority on how any conversation should include this service. Sometimes, due to internal policy evolution, the owner of a particular service may wish to change its conversation model, e.g., change the preconditions for its invocations. Such a change would imply, as a side effect, that previously composed processes involving the service in question now communicate with it in an illegal manner (as per the new model) and thus generate exceptions or cause undefined behavior.

3. The WRABBIT Workflow Reconfiguration Architecture

Our approach to conversation failure recovery through workflow resynchronization is implemented in the "Workflow Reconfiguration with Agent- and BPEL-Based Intercommunication Technology" (WRABBIT) framework. Figure 1 illustrates the software architecture of each WRABBIT agent.

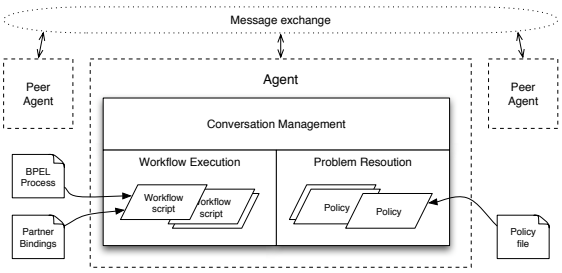

Figure 1: The agent architecture

As noted above, the execution of a workflow is directly linked to communication between the partners in that workflow. As in BPEL, the work that falls outside the agent's conversation-management role is accomplished by traditional web services. Inter-agent conversations, on the other hand, use an ACL[1]. For this work, we have adapted a subset of ACL messaging primitives defined by FIPA [8]. FIPA also provides an XML encoding for ACL messages, allowing these to be exchanged using web-service technologies. Some middleware for web-service deployment and execution is thus required by the WRABBIT architecture, and we have selected AXIS [2] to fill this need.

On top of this core layer rest a suite of capabilities enabling the agent to

1. interpret BPEL process specifications and use them to construct workflow scripts, from which it extracts conversation scripts that define message sequences with its partners;
2. execute these workflow scripts at run time;
3. recognize conversation failures from discrepancies between the sequence of ACL messages it expected according to its conversation script and the actual sequence of ACL messages it has received, and making the other agents in the conversation aware of the failure

[1] We use the term agent communication language (ACL) in a generic sense to denote a small set of high-level message primitives, described below, and not a full implementation of an ACL standard, such as [8], [11]

4. diagnose the cause of the failure as a fault in the workflow script followed by itself or the other agents;
5. engage in conversations to obtain the correct workflow script, and
6. adapt its state as appropriate.

The remainder of this section provides implementation details on these capabilities.

3.1. BPEL-to-ACL translation

As noted earlier, our fundamental methodological assumption is that distributed workflow orchestration requires a conversation among a set of collaborating agents, where each agent is capable of delivering some of the services included in the workflow. We have defined a method for translating a BPEL process into a conversation script that specifies the expected ACL message exchange sequences.

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-ordermessage primitives. 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 [8]. *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-type, and operations 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. The mapping is summarized in Table 1.

Consider the case of a synchronous BPEL invoke activity (the type that invokes a WSDL *request-response* operation, as described in section 11.3 of the BPEL specification [3]). In this interaction, the invoker initiates the exchange 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. This exchange is translated into the following message sequence: the requester sends an *inform* message (with the WSDL input message as content) immediately followed by a *request* message (requesting the WSDL output message) and the receiver sends an *inform* message (with this WSDL message as content) as a reply. We interpret any message sent between web services as information of one kind or another, and thus for these we use *inform*. In this example, the invoker of the operation is actively soliciting a response, as is clear from the semantics of the WSDL request-response operation, and thus a *request* is also part of the communication model.

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

BPEL Activity	***Message Sequence***
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 from BPEL activity to ACL message sequence

3.2. Workflow Reconfiguration Support

The agent's ability to dynamically re-synchronize workflow models relies on the abilities to (a) initiate or respond to conversations with other agents, (b) recognize and relate conversation failures to workflow mismatches, and (c) compose simple workflow scripts

3.2.1. Conversation Layer

Each agent participating in a workflow composition reads in the BPEL process specification corresponding to its role and creates a workflow script. If an agent participates in more than one workflow, then it has a corresponding workflow script for each one. At run time, as soon as the agent receives a message initiating a particular type of conversation, it instantiates an instance of the workflow script that encompasses the corresponding conversation script. It uses this instance to formulate expectations for subsequent message exchanges. Each script serves as a detailed protocol for an interacting agent that specifies not only the order in which the ACL message types can occur, but also their required content. The conversation script elements enable the agent

to formulate expectations of what messages these agents will be sending in the future, until the conversation concludes.

WRABBIT agents are designed to achieve objectives of two general types: objectives to bring about a state *s* and objectives to execute action *a*. Workflow script instantiation is an example of the latter type. The algorithm that WRABBIT agents use to process their objectives is outlined in Figure 2.

3.2.2. Conversation Errors and Correction Policies

Consider the case where a workflow is unilaterally modified, causing the participating agents' workflow scripts (and thus conversation scripts) to become out-of-sync. Suppose that Organization A adopts a new policy, adding to the preconditions that must be satisfied for it to contribute its services to the 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, still has the old BPEL process x_{client} and uses it as the basis for constructing its respective workflow script. As a result, Organization B's agent will invoke Organization A's service, sending the initial message without the necessary precondition being satisfied. From Organization A's perspective, this message is "unexpected." The agent perceives such failures as symptoms of out-of-sync conversation models. The discrepancies can be recognized as being in one of the following categories:

1. message *m*'s content invalid, i.e., the message payload is foreign to the receiving agent;
2. message *m*, received from the sender, constitutes an illegal interaction (the receiving agent's conversation script does not include the receipt of this message from this particular sender);
3. message *m* is out-of-order (according to its conversation script, the receiving agent is not expecting this message at this time).

Any one of these failures triggers a *not-understood* message, sent by the receiver of *m* to the sender of *m*. The

Agent objective achievement

Until all objectives are achieved, for each unachieved objective **o**:

1. If objective **o** is an objective to determine the value of an information type *i*, then:
    a. call the composition algorithm with information-type *i* as input to generate a new composed workflow-script
    b. create a new objective to execute the script
2. else if objective **o** is an objective to pursue the execution of a workflow-script, then:
    a. if the script requires a message in an open conversation, and the objective does not have the message, then try again later
    b. else continue the execution of the script
3. else if the objective **o** is an objective to route incoming messages to their destinations, and a new message m has been received then:
    a. if the message *m* is associated with open conversation *c* and is a *not-understood* message, then *resolve conversation failure*
    b. else if message *m* is associated with open conversation *c* and is legal/expected, process the message within the objective that is executing the workflow script that defines *c*
    c. else if message *m* starts a new legal conversation that is defined by some workflow script *w* that is provided by this agent, then set an objective to execute *w*
    d. else if message *m* does not legally continue an open conversation with this partner or does not legally start a new conversation with this partner, then
        i. generate a *not-understood* as reply to message *m* with the appropriate reason
        ii. *resolve conversation failure*

Resolve conversation failure

1. interpret the "error reason" contained in the *not-understood* message or obtained during detection
2. identify the policy for dealing with this type of error
3. determine if the policy requires changing the workflow scripts of self or of some other agent *x*
4. if the source for the correct workflow scripts is self, then no further action necessary
5. else if the source of correct workflow scripts is agent *x*, then request new scripts from agent *x*

Figure 2: WRABBIT agent execution algorithm

content of this *not-understood* includes a 'reason' that corresponds to one of these three failure categories. The generation and receipt of a *not-understood* starts an exception-handling conversation, in which the two agents exchange messages that may allow them to recover from the failure. The objective of this conversation is to identify a possible adaptation of their corresponding conversation scripts, which would (a) be acceptable according to the agents' shared policies, and (b) eliminate the discrepancy that caused the failure. Simply put, one of the two interacting agents has to change its conversation script to fit the other's. The question is, whose conversation script will be used?

In principle, there may exist complex policies for determining the answer. Some may depend on detailed contractual agreements between the conversing agents' organizations (for example, an organization might be allowed to change the workflow only during a pre-determined range of time, perhaps corresponding to off-peak hours); others may depend simply on "authorities" assigned for each particular conversation. Other policies might associate authorities for different sorts of functionalities. Agents have access to a declarative policy that states which agent serves as the authority on the currently legal conversation model governing some particular conversation.

In our example, Organization A is the authority for this workflow interaction. Its agent detects the failure, and then would generate the *not-understood* message described above. It would consult the shared declarative policy, and determine that it has the correct model (which was derived from x'). The receiver of the *not-understood*, Organization B's agent, consults the same policy and identifies that the sender is the authority for the current conversation. It then initiates a conversation with the authority agent, with the intent of obtaining the up-to-date workflow script, based on the BPEL process x'_{client}, that corresponds to its role as client. This engages the third key element of the agent layer, a simple workflow script composition algorithm.

3.2.3. Workflow Script Composition

While BPEL can be used to specify a complete, executable implementation of any given process, it can also be used to define abstract processes that leave some details unspecified. We assume that an agent (corresponding to an organization participating in the workflow) would use a complete BPEL process to control its own operation, and would specify how other organizations should interact with it using an abstract BPEL process. WRABBIT agents exchange abstract workflow scripts based on these abstract processes to repair their operation. These are then used together with a simple workflow composition algorithm to create executable workflow scripts (see Figure 3).

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 (WSDL-level) message types. In response to a need for the value of a particular message, the algorithm searches the available workflow scripts for one that satisfies this need. If the selected script itself is abstract and needs additional values, the algorithm seeks these out in the same manner. The selected workflow scripts are ordered such that their dependencies are satisfied, and the composed workflow script can then be executed to obtain the initially needed value.

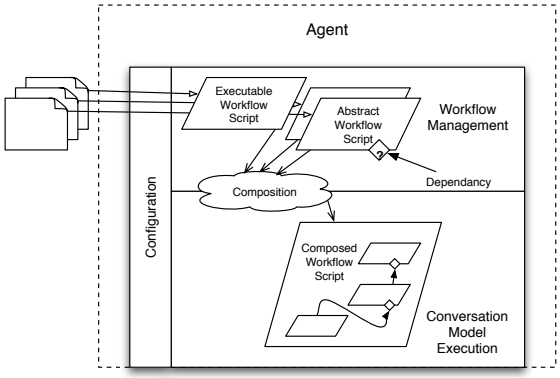

Figure 3: Conversation model composition

4. Case Studies

Our case studies are inspired from intuitive scenarios of workflow reconfiguration as it often happens in academic departments. Suppose that various people in a department have wisely automated some of their day-to-day activities as web-services. In particular, the department's administration has a service that provides student transcripts to faculty members, one that allows modifications to student grades, and another that provides the electronic equivalent of stamping authorization forms for faculty members. The faculty members use these services in workflows such as displaying a student's transcript, or correcting mistyped grades.

The department's student transcript service will be the focus of our case studies. It is built using WSDL one-way operations: it first requires a message containing the student's id and the address of a callback function, to which it then provides a message containing the student transcript. However, in response to new privacy legislation, we suppose that the department now requires that all access to student records be authorized. To enforce

this, the service is changed to require a new initial message that contains an authorization descriptor.

The following case studies demonstrate how an agent that encapsulates a faculty member's workflows will adapt to the new authorization requirement. The execution details were gathered by examining the log files of the WRABBIT agents after each scenario's execution.

4.1. Case Study 1

The first case study examines what happens when the instructor simply wishes to display a student's transcript. The messages exchanged during this scenario are depicted in Figure 4.

In this case, the instructor's WRABBIT agent is made to locate or compose a workflow script that ends in the production of a student transcript message. Since the workflow script that retrieves the transcript from the department's agent produces this message, the system selects the script for execution. However, as the department's agent has been configured with the updated workflow script that requires the authorization token, the initial message from the instructor agent causes a conversation failure (Figure 4(a)). A sequence diagram of how this initial message is handled is shown in Figure 5.

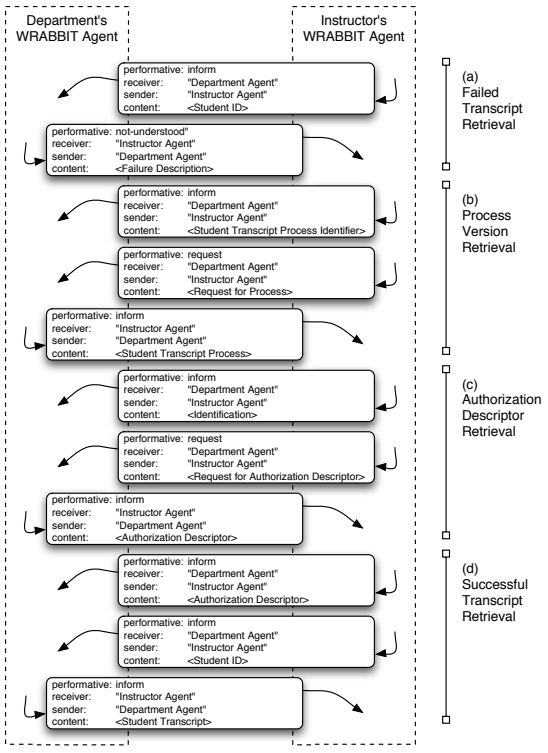

Figure 4: Messages Exchanged

The department's agent sends a *not-understood* message to the instructor's agent, and terminates the execution of the script. Upon receipt of the *not-understood* message, the instructor's agent knows that the conversation associated with its "Student Transcript Retrieval" workflow script has failed (step 3a of Figure 2). Using the failure reason included in the content of the *not-understood* message, the agent selects the workflow authority policy to use, which in this case identifies the department's agent as the authority for this workflow script. The instructor's agent initiates a sub-conversation with the department's agent to obtain an updated workflow script (Figure 4(b)).

Once this is done, the instructor's agent once again composes a workflow script that produces a student transcript (step 1 of Figure 2). The updated "Student Transcript Retrieval" workflow script is abstract (i.e. has a dependency), in that it does not specify how to obtain the authorization descriptor. For this scenario, we have provided the instructor's agent with the workflow script for obtaining an authorization descriptor from the department's agent. The composition algorithm includes this script in the composite workflow script, and is executed first (Figure 4(c)). The conversation is reinitiated by the instructor agent and is now successful. (Figure 4(d)).

4.2. Case Study 2

In the second case study, we demonstrate an optimizing feature of a WRABBIT agent's execution process. When a workflow script is composed from multiple sub-scripts, and a conversation failure occurs during the execution of one of these scripts, only changed or added sub-scripts are re-executed after the recovery procedure.

In this case study, in order to satisfy its needs, the instructor initially needs both a student's transcript and an authorization token. However, as before, the transcript service as known to the instructor agent does not require the authorization service. Rather, the authorization token is required separately. Therefore, the instructor's agent composes a workflow that produces both an authorization descriptor and a student transcript. This results in a composite workflow script that features both the original "Student Transcript Retrieval" workflow script and the "Authorization Description Retrieval" workflow script.

Once the "Authorization Description Retrieval" workflow script is executed and the authorization descriptor obtained, the "Student Transcript Retrieval" workflow script is attempted, but is subject to the same conversation failure as in the previous case study. As before, the instructor agent obtains the new "Student Transcript Retrieval" workflow script and constructs a new composite workflow. However, the authorization script has already been executed successfully, and it has not changed. Therefore, the WRABBIT agent simply uses

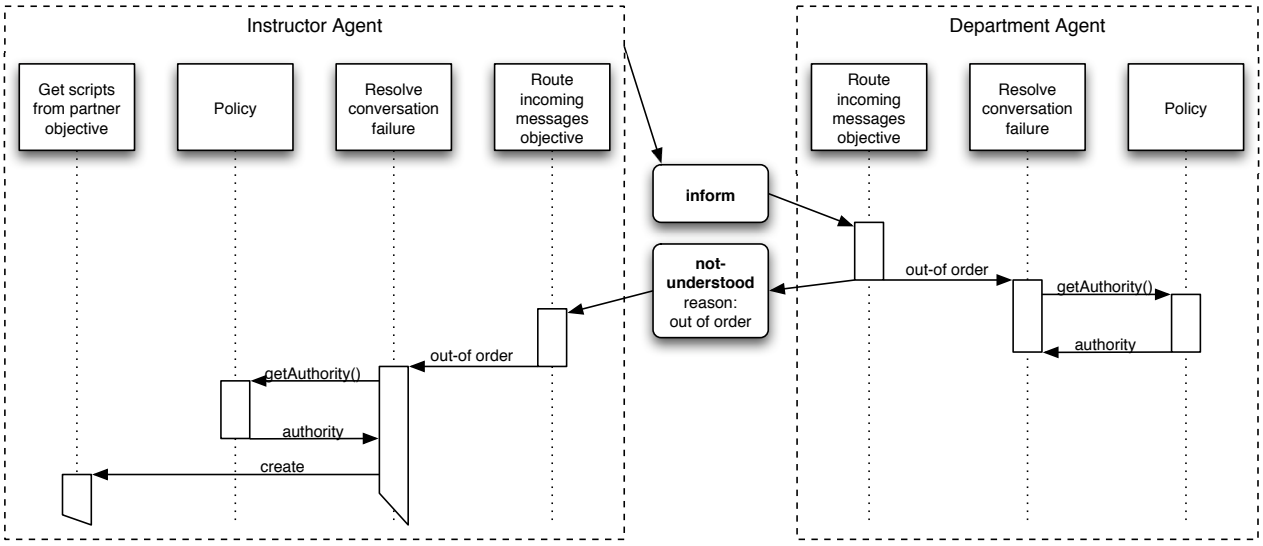

Figure 5: WRABBIT message processing

the result of the previous execution rather than re-executing that sub-script. This limits the impact of workflow script changes caused by conversation failure.

5. Related Work

Web services composition is an area of active research. The community is still debating the issue of centralized vs. distributed coordination of web-service compositions [15].

Although most mature middleware support for BPEL process execution assumes a centralized view, distributed environments are receiving increasing attention. The Symphony project ([6], [13]) has developed an algorithm for analyzing a composite service specification for data and control dependences and partitioning it into a set of smaller components. 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.

Blake has also conducted extensive research on how agents could be used to support better workflow execution. 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.

In the general distributed workflow execution and management area — outside web services specifications — there has been a lot of work on workflow change management. We mention [10] as a representative example. They present 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 model, 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. Some frameworks dynamically combine small protocol units as a way to respond to unexpected messages (e.g., [12], [14]). This approach has agents engage in *query*, *inform*, or *error* sub-conversations whenever necessary, returning control to the 'main' conversation protocol as required. Other research [7] derives the definition of normative communication from an underlying distributed task model from a more abstract normative communication model, a concept this work has extended by adding error resolution.

6. Conclusions and Future Work

This paper has presented the WRABBIT framework, which examines a plausible solution to the problem of adaptively maintaining workflows in the face of the independent evolution of their constituent workflow scripts. These scripts are derived from BPEL processes,

which are used for orchestration in the web-services arena.

When one agent's workflow changes unilaterally, it may incur conversation errors with other agents. We presented a small set of such error types, such as formerly required information that is no longer necessary, re-ordered steps, or new preconditions. The symptom is an illegal message at run time, which is identified using *not-understood* messages. To date we have focused on "out of order" illegal messages, caused by the addition of newly required message exchanges introduced at the beginning of the agent's conversation model.

In response to a *not-understood* message, the agents engage in a new conversation, where the authority agent – as defined by the shared failure-recovery policy of the workflow – dictates to the other agents the current, correct workflow script. The subordinate agents adapt their own workflows by reorganizing them so that they meet the constraints of the prescribed workflow script. A further benefit of this behavior is isolation of the recovery effort. The error is detected and resolved between the two partners; other partners are not affected. This will support distributed management, where the owners of smaller workflows can evolve these fluidly, without concerning themselves with the orchestrated higher-level workflows in which their smaller workflows are components. Our ongoing efforts are directed at exercising our framework on more complex workflow changes and policies that govern the ensuing adaptations, and ensuring that the key properties of the original workflow continue to be preserved in the adaptations.

References

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

[2] Apache Axis http://ws.apache.org/axis/

[3] 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).

[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] G. Chafle, S. Chandra, V. Mann and M. Nanda. "Decentralized Orchestration of Composite Web Services." In *Proceedings of the Alternate Track on Web Services at the 13th International World Wide Web Conference (WWW 2004)*, New York, NY, May 2004.

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

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

[9] 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.

[10] M. Kamath and K. Ramamritham. "Pragmatic Issues in Coordinated Execution and Failure Handling of Workflows in Distributed Workflow Control Architectures." Univ. of Massachusetts Computer Science Technical Report 98-28, August 1998.

[11] 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.

[12] S. Moore. "On conversation policies and the need for exceptions." In *Issues in Agent Communication (LNAI 1916). Edited by* F. Dignum and M. Greaves. Springer-Verlag, Berlin. 2000. pp. 144-159.

[13] 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.

[14] 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.

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

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