University of Alberta

Enabling Automatic Recovery from Communication Failures between Composed Web Services

by

Warren Scott Knight Blanchet ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of *Master of Science*

Department of Computing Science

Edmonton, Alberta

Spring 2006

Your file Votre référence
ISBN: 0-494-13795-9
Our file Notre référence
ISBN: 0-494-13795-9

Dedication

To Aunt Edith, who encouraged all my academic endeavours.

Abstract

An organization's systems must respond to changes in the processes they automate. When an organization delivers web services for composition into one or more inter-organizational workflows, their independent evolution can cause problems. Specifically, when previously expected messages between a service and its partners become unexpected due to that service's evolution, the distributed service workflow can fail. This occurs because the respective workflow models of the components are no longer synchronized.

This work presents an intelligent-agent conversation framework with which web services are implemented. The system can adapt web service workflows in response to failures caused by out-of-sync workflow models through the use of a globally shared failure recovery policy. This policy allows agents to resolve various types of model mismatches that cause interaction errors, including changes to required preconditions, partners, and expected message ordering. Case studies are also presented that illustrate the approach of the framework and its assumptions.

Acknowledgements

I thank my supervisors, Drs. Renée Elio and Eleni Stroulia, for their invaluable guidance during my time as a graduate student. Their knowledge and experience were crucial resources, and their assistance was beneficial in innumerable respects.

I also thank the members of the Software Engineering research group who, over the course of my years as both a graduate and undergraduate at this university, have stimulated my interest in research. Many of them have offered feedback, assistance, and advice, all of which have made my tenure here enjoyable and rewarding. I would like to single out Stella Luk, who worked on the project during the summer of 2005, for asking so many questions; my attempts to answer them uncovered conceptual defects that I would not have found on my own.

I thank my friends and family. While they did not contribute a sentence to this thesis or a line of code to the implementation, I am confident that both would have suffered should they have been absent.

Finally, I thank my wife, Julie, for the million little things she does to make my world a better place in which to live.

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction

1.1. Evolution of Web Service Compositons

Web services are an important development in distributed computing. Their promise of implementation independence has been substantiated, with web service technology already enabling many organizations not only to integrate their existing applications, but also to interoperate with other organizations.

However, some aspects of web services and the related standards that define them are still the subject of non-trivial revision, addition and replacement, with efforts merging or diverging as dictated by market, academic and political forces. One aspect in particular that is seeing much activity is the composition of web services. For the many organizations that subscribe to the Enterprise Service Bus (ESB) architectural model, in which a large set of basic, stateless services are used by larger enterprise applications, these efforts are particularly interesting. If standards and supporting technology for composing services together were available and reliable, then organizations could make better use of their ESB services by using this technology to build their enterprise-level applications.

For example, consider a business that ships goods for partner businesses. This business's ESB would likely feature services that interact with the various databases and systems in the organization, while the enterprise-level applications would use these services to accomplish larger units of work. Consider this business's deployment of web services, depicted in Figure 1.1. Here, lower-level ESB services that bill the customer, create shipping records in a database, and remotely dispatch an employee to pick up the shipment are composed together to provide a higher-level shipping service to the business's customer, so that it can integrate shipping into its workflow. As integration between organizations becomes more common, the importance of these compositions will increase, which increases the value of tools and standards that facilitate their construction.

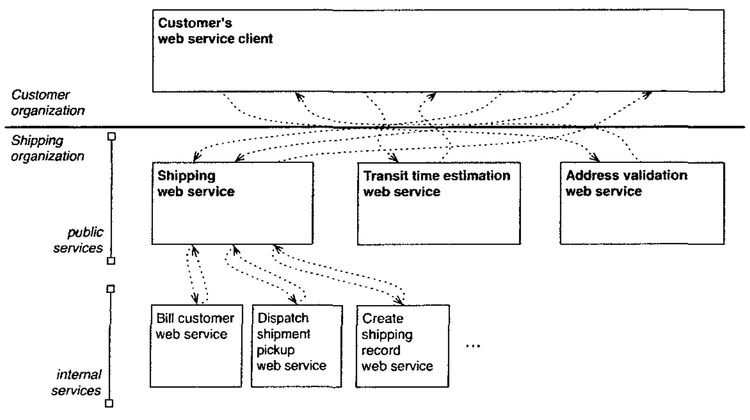

Customer's web service client
Customer organization
Shipping organization
Shipping web service
Transit time estimation web service
Address validation web service
public services
Bill customer web service
Dispatch shipment pickup web service
Create shipping record web service
...
internal services

Figure 1.1. Example Web Service Deployment

One such service composition standard is BPEL4WS (often shortened to BPEL), the Business Process Execution Language for Web Services [BPEL4WS]. It is used to both model and implement (using already-available BPEL execution engines) one web service endpoint through the use of a process-focussed task model, where the tasks are limited to data manipulation and web service-based message exchange. One possible use of BPEL would be to construct a new application by composing a group of simple web services, perhaps taken from an organization's ESB, and subsequently to deploy it through the use of a BPEL execution engine. The result, however, would be a heavily centralized application implementation, which is unrealistic in most cases. In order to be useful, the approach used must model the complex interactions that large applications have with their peers, or alternatively in the case of a distributed application, that a component has with its other components. Thus, web service endpoints implemented in BPEL are likely to include message exchanges, implemented using web services and structured using BPEL's process-description facilities, that are to be used as part of interaction protocols with other web services.

While constructing the composed services is a critical step, they, like any other software, will have to be maintained. An important component of this maintenance is modification in response to changes in the business processes implemented in the compositions. In the shipping business example, consider the effect of the business expanding into another country, and providing international shipping. Previously, it provided only national shipping, and thus the country portion of any address used in the services was implicit. The company's new customers will undoubtedly wish to integrate its shipping services into their workflows as well, so those services will have to be modified to explicitly include the country. However, what effect will this have on existing customers? Their services' actions depend on the unmodified service's behaviour.

Such composition evolution is not yet well handled by web service standards. While BPEL may be used to implement suitably complex applications involving a set of peers, each BPEL specification models only one such peer, and at the time of writing there is no finalized and well-adopted standard or technique for specifying the corresponding complex inter-endpoint message exchange protocols from a *global* viewpoint. Such protocols are modelled only in a distributed manner as the collection of per-participant BPEL specifications. Thus, this standard addresses neither the challenge of verifying the interoperability of a set of interacting endpoints, nor the challenge of modifying the behaviour of interoperable endpoints with minimal difficulty. The challenge of verification is already addressed by developing standards and research further elaborated upon in Chapter 2, *Problem Context,* while the second challenge is the focus of this work. Because BPEL specifications are peer-specific, if an endpoint's BPEL specification is modified to include a new message exchange, the specification of the peer's behaviour will also have to be modified to include the new message exchanges. These new specifications will then have to be deployed in some way that guarantees that both endpoints will continue to interoperate. If during deployment, one endpoint uses the modified specification while the other uses the unmodified one, a message that is unexpected by one of the two peers may be sent by the other, causing incorrect or undefined behaviour. This work presents a solution to this deployment problem that is triggered by these invalid message exchanges, enabling such applications to repair themselves autonomically at run-time under certain circumstances.

1.2. Chosen Approach

Relevant experience useful in addressing this issue can be found within the agent communication community. This collection of research is focussed on the problems that intelligent agents face when the achievement of their goals requires communication with one another. Successful communication between two parties always involves a shared understanding as to what communicative acts to expect and as to the meaning of the content of these communicative acts. Since agents exchange messages in the pursuit of some goal, the messages exchanged between agents are grouped together into goal-related conversations. Given this terminology, the specific contents of a normative communication model can be defined by "conversation policies", as introduced by [GHB00]. Such policies were advocated to be publicly available declarative specifications of constraints on the otherwise unlimited number of possible message sequences and content that could appear in any given conversation. The arrival of a message that violates some constraint of any of the conversation policies currently used by the agent is considered to be an exceptional event, which causes the conversation to fail.

This work adopts the approach of [EP05], in which the task model is used to define a normative conversation model. Given the current context of the task's execution, the roles and responsibilities that each agent is known to have for this task's execution, and so forth, an agent using this approach is thus able to determine what messages are legal or expected. Since tasks in a web service application can be modelled using BPEL's process-based service composition facilities, BPEL specifications could serve as the basis of such a normative communication model through the use of this approach. Once a normative model is obtained, it can then be used to define classes of errors associated with particular violations of the model's constraints, identifying the failure caused by a particular message with greater specificity. The occurrence of these failures indicates that the underlying declarative specifications that govern the agent's behaviour do not match, as otherwise the unexpected message would not be sent.

Assuming such a model for defining conversation failures, a BPEL execution engine could detect conversation failures at run-time, providing the application developer with improved diagnostic materials. However, the agent communication community has more to offer. First, the agent communication languages (ACLs) used by the communicative-agent community define message primitives to signal failures of understanding in general, of which conversation policy violations constitute a subset. The two ACLs of note are the Knowledge Query and Manipulation Language (KQML) [LF97], and the Foundation for Intelligent Physical Agents' (FIPA) ACL [FIPAActs]. FIPA ACL's failure-of-understanding primitive message label (or "performative", as they are called in the community) is *not-understood*, while in KQML, the performative is *error*. Similar messages could be exchanged between BPEL execution engines, signalling conversation errors as the ACL messages can be used between agents. Second, conversation policies were also presented as an exception-handling mechanism. If BPEL execution engines adopted a shared policy for handling conversation errors as the engines recognized them or as they received messages signalling their occurrence, the policy could include actions to take in response to particular conversation errors that would prevent these from reoccurring. Since these failures indicate mismatches between the endpoints' models, this policy would have to impose an authoritative set of matching models on both agents, using the particular failure type and other information to locate the authoritative models.

This work builds upon previous work in the agent communication community, both on the definition, detection, and signalling of communication failures, and on exception recovery, by proposing a method through which such communication failures could be resolved using a shared policy. The developed method is applied in the realm of web service composition, specifically to compositions modelled and implemented using the BPEL specification. Conversation failures are introduced into this realm through unilateral modifications of the modelled process by a participating endpoint. These endpoints are each implemented using a BPEL execution engine that includes agent-based technology, allowing the endpoints to repair their mismatched behaviour specifications. Several case studies are presented to illustrate the problem and the implemented solution.

1.3. Thesis Organization

This thesis is organized as follows. Chapter 2, *Problem Context*, explores the various literature related to this work. Specifically, various web service specifications are covered, to provide the reader with the background necessary to understand the domain in which this work is applied. Then, various research efforts in the web services arena are discussed, with an emphasis on work that addresses related problems. This is followed by an overview of the agent communication research whose ideas and constructs are integrated into this work. Having established this foundation, the chapter proceeds to illustrate the problem that this work addresses in more detail, and outlines the strategy used to address it. Finally, work featuring similar strategies is compared and contrasted with this work.

Chapter 3, *Solution Implementation* follows, and provides a detailed examination of the proposed solution to the problem. The solution was implemented as part of this work, and a comprehensive inspection of this software is also included in this chapter. Chapter 4, *Solution Evaluation*, includes a description of the case studies that were performed as part of the evaluation of this work. Each case study features a scenario in which a distributed web service workflow is modified such that the composed endpoints encounter communication failures. The studies then explain how the proposed mechanism is able to resolve the failures and re-execute the workflow successfully.

Finally, Chapter 5, *Conclusion*, identifies the contributions of this work, and discusses the limitations of both the proposed solution and its current implementation.

Chapter 2. Problem Context

Contemporary enterprise-level software systems tend to be distributed, composed of heterogeneous technology, and process-oriented. Researchers have been wrestling with the challenges posed by the construction, maintenance and evolution of such systems for some time, and many communities of research have organized around particular approaches to addressing them. This work draws upon two such communities, the web services community and the agent communication community. Recently, web services have become an increasingly popular mechanism to address the challenge of heterogeneous implementation technology, both in industry and in academia. Web services intend to replicate the success of the world wide web, through the use of standards for data encoding, data exchange, and other concerns, each independently implementable by disparate groups. An overview of these standards and related research is provided in Section 2.1, "Web Services". Similarly, the agent communication community coalesced to address the issues associated with distributed systems. Their particular focus is the distribution of capabilities and knowledge among a group of intelligent agents. Relevant work from the agent communication area is discussed in Section 2.2, "Agent Communication Research". These two bodies of research have helped to shape the scope of the problem addressed by this work, and the direction chosen to solve it. This influence is explained in Section 2.3, "Problem Scope". Finally, the particular approach of this work can be compared with aspects of work previously accomplished. That related work is identified and contrasted with this work in Section 2.4, "Related Work".

2.1. Web Services

Web services seek to bring the reach and interoperability of existing internet-based user-application interaction to its inter-application complement. The world wide web has fundamentally changed how many organizations operate, with the greatest area of impact being their interactions with customers. The web allows customers to access information and to conduct transactions at any time, from any place. Organizations benefit from the increased automation that is possible while using the web, allowing them in turn to improve their offerings to customers, attracting more business. However, current web technology is focussed on human to web-application interaction. Rendered HTML relies on the human mind's excellent image processing capabilities to segment and comprehend the displayed content; computers have no inherent abilities in this arena. Therefore, to foster the same benefits the web offers to inter-application interaction, the content of the exchanged information must be made machine-understandable. One approach is to design the exchange with machine-understandability as a requirement.

With this goal in mind, many organizations have published specifications describing models and syntax for specifying some aspect of such interactions. Collectively, these specifications have become known as the "Web Services Stack". Some of these efforts have been successful, and enjoy high visibility and good implementation support. New entrants are common, and offer improved and interesting perspectives within their chosen scope. Web service specifications share some common features, particularly the use of XML and existing web communication technologies. The use of XML brings the benefits of standard parsers, satisfying some of the requirements for machine-understandability. The use of existing web communication technologies such as HTTP and SMTP ensures that existing network configurations will not have to be modified to accommodate the new paradigm. Section 2.1.1, "Web Service Stack Specifications", covers some relevant members of this stack.

The research community has also been active in the web services arena, addressing those areas not yet covered by web service standards. Some of the most relevant work is covered in Section 2.1.2, "Web Service Stack Research".

2.1.1. Web Service Stack Specifications

A model for the use of web services evolved alongside the web service stack's first components. This model, as it is described in [Kre01], features three roles: a service requester, a service provider, and a service registry. Operations connect each of these roles to each other. The service provider first *publishes* descriptions of its services by communicating with the service registry. The service requester then *finds* the services that it needs by communicating once again with the service registry. The service requester and service provider then *bind* so that the requester's implementation may access the service provider's service implementation. The model is diagrammed in Figure 2.1, "Web Service Usage Model", which was adapted from Figure 1 of [Kre01].

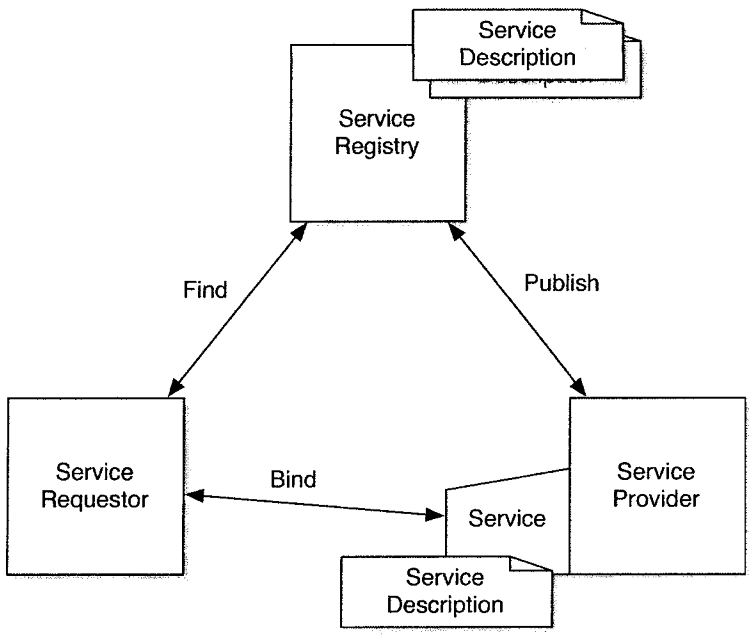


Figure 2.1. Web Service Usage Model

Early web services efforts focussed on the protocols and specification languages required to enable these three operations. Of primary importance to this work is the Web Service Description Language (WSDL). This specification fills the need for describing web services, a necessary precondition to communication about services. It is discussed together with its companion specifications in Section 2.1.1.1, "Web Service Description". Later work has been directed in part at specifying various aspects of the behaviour of each of the roles. Some effort has been spent on developing protocols that enable various web services to coordinate the execution of a unit of work, for example to maintain transactional integrity. These standards are examined in Section 2.1.1.2, "Web Service Coordination". Other work has examined how web services could be used in workflows, both as units composed together to accomplish work and as a message exchange mechanism between workflow endpoints. Section 2.1.1.3, "Web Service Composition", covers this material.

2.1.1.1. Web Service Description

Specifications written in the Web Service Description Language, or WSDL, describe the interface and the location of software services [WSDL]. A WSDL interface consists of simple exchanges of structured messages between a service provider and its clients. The messages exchanged are XML documents, optionally conforming to a specified XML–Schema. The message exchanges themselves are called *operations*, and WSDL defines four types:

One-way	The service provider receives a message of a specified type.
Request-response	The service provider receives a message of a specified type, and responds with one of a set of typed messages, all but one of which signal an exceptional condition (which WSDL calls a *fault*).
Solicit-response	The complement to a request-response operation: the endpoint sends a message, and receives one of a set of possible messages.
Notification	The complement to a one-way operation: the endpoint sends a message.

WSDL additionally provides support for defining groups of operations, which it calls *port types*.

In order for a WSDL specification to be useful for locating a particular service and binding to it, the operations described must be associated with a client implementation. WSDL supports this through the instantiation of its *port type* specifications, accomplished using its *port* specifications. One important characteristic that these specifications need to describe is the mechanism used for message exchange, and WSDL provides an extension facility for this purpose. The specification also provides a default set of extensions for describing message exchange mechanisms based on the use of SOAP (the Simple Object Access Protocol, [SOAP]), or based on the use of HTTP GET and POST operations.

WSDL has been widely adopted and implemented, and with greater interest comes greater scrutiny. While work on the next revision of WSDL is ongoing at the W3C, the Web Services Interoperability Organization (WS-I) has published several documents, which it calls profiles, whose aim is to clarify or eliminate ambiguous elements of various web standards. An implementation can claim conformance to a WS-I profile if it satisfies the numbered requirements that it contains. This allows for interoperable implementations of existing standards, an important requirement since these implementations are being deployed now, and thus will not benefit from any improvements present in the next revision of any standard.

One of the WS-I profiles is the Basic Profile [WS-IBasicProfile], whose primary focus is on SOAP and WSDL. The Basic Profile eliminates the *solicit-response* and *notification* WSDL operation types, as these are "not well-defined" (§ 4.5.2). Indeed, the WSDL specification does not include any material on how these message exchange types would be used or implemented, and its included extensions for specifying message exchange mechanisms do not cover these operation types. Furthermore, this deletion is merely an affirmation of the use case for WSDL specifications chosen by an overriding majority of the web services community: describing web services from the perspective of their implementation. In effect, only the *one-way* and *request-response* operation types are significant, and therefore only these will be discussed in the remainder of this work.

Nevertheless, WSDL's operations and ports satisfy the web service model's requirement for a standard format for describing the message signature and implementation location. Web service requesters require the typed structure for message content and the operations in which these messages are exchanged to deploy a compatible service requester implementation. The requester implementation in turn requires the information necessary to locate and communicate with the implementation of the web service provider. Also, because WSDL is extensible, it can accommodate both the web service description needs that currently exist and also any new specification methods developed to fill needs not yet identified at the time it was written. Indeed, for some applications, the level of web service description provided by WSDL alone is insufficient. For example, policies on the use of the web services are not describable using standard WSDL.

To address this need, WS-Policy was proposed [WS-Policy]. This specification provides a framework in which policies on web services can be defined. Specifications that describe some aspect of a web service's operational capabilities or requirements can thus define their model and syntax within the constraints defined by WS-Policy, and then benefit from its mechanisms for policy selection and management. For example, [WS-SecurityPolicy] defines many aspects of a web service's security using WS-Policy: what message encryption method is required, what message integrity verification method to use, etc. A service requester seeking to bind to a service provider would have to conform to this policy in order to ensure interoperability. The method by which policies defined in WS-Policy are associated with specific web services is described in a separate specification, [WS-PolicyAttachment]. Fortunately, because of WSDL's extensibility mechanism, the attachment mechanisms defined in that specification are quite straightforward, allowing policies to become integrated with the rest of the web service description.

2.1.1.2. Web Service Coordination

One particularly interesting set of policy specifications for web services are the web service coordination standards. The aim of these policies is to provide support to web services that operate as a component of a distributed application. To satisfy this goal, these standards provide protocols that web services may use to coordinate their actions with other web services. A web service that includes such a policy indicates that any potential web service client must be capable of following such a protocol to satisfy the requirements of using the service. It is important to note that the protocol used is separate from the messages exchanged between the two participants, and any ordering that may exist on these. To clarify, an overview of these coordination standards follows.

The basic standard in this group is WS-Coordination [WS-Coordination], which defines a framework for coordination protocols. The model proposed in the standard features a set of coordinator endpoints, each featuring services for activation and registration. These coordinator services are used by the coordinating services to support the execution of distributed applications. First, as it begins a coordinated activity, a web service will use the coordinator activation service. This activation service returns a coordination context, which the web service then distributes to its partners in the distributed activity. Such context distribution occurs within the application-defined message exchanges between the partners. Having received the coordination context, these partners then use the coordinator's registration service to register themselves as participants in the coordinated activity, providing the context they received. The model allows for a federated network of coordinator endpoints, allowing organizations to distribute the coordination work by directing coordinating services to use different endpoints. However, the coordinators themselves must still communicate between each other, and thus considered as a set are a centralized solution to the coordination issues facing distributed applications.

There are two widely known coordination protocols: WS-AtomicTransaction [WS-AtomicTransaction] is used to coordinate atomic transactions of a short duration, and WS-BusinessActivity [WS-BusinessActivity] is used for longer-running business transactions. As its name implies, WS-AtomicTransaction implements atomic transactions, specifically through the use of two-phase commit coordination protocols. Coordinating web service endpoints may register themselves as volatile or durable participants, depending on the type of the resources they manage (for example, a cache is a volatile resource, while a database is a durable resource). Additionally, there is a protocol to begin the two-phase commit process. The other standard, WS-BusinessActivity, is for use in situations where the distributed activity may take a long time, for example, it might have to wait for human approval or even product delivery. As such, atomic transactions are not suitable, and therefore the standard relies on the participants to provide compensation behaviour in the case of failure. Its two protocols allow the coordinator to update all participants with the current state in a predefined state space describing the status of the coordinated activity, with the distinction between the protocols being whether the initiating participant or the coordinator is tasked with determining the end of the business activity and thus transitioning to the completed state.

Coordination allows web service endpoints to agree on the state of their shared activity, which is an important feature required in many contexts. The approach of WS-Coordination is to centralize authority of the subset of this state necessary for agreement on the outcome of the distributed activity. This leaves the remaining state of the shared activity to be managed in a distributed manner by the activity's participants.

As such, the use of WS-Coordination does not provide coordination between web service endpoints, but only support for such coordination. A separate set of standards addresses the issues that arise when multiple web services are used in concert, or "composed", to become new applications.

2.1.1.3. Web Service Composition

The use of web services as the building blocks with which larger applications are built is a compelling idea. Creating new services by composing together existing services brings the benefits of modularity to the web services arena. In a particularly good survey of web service composition efforts [Pel03], Peltz makes an important distinction between two different perspectives on composition: orchestration and choreography. Orchestration is the perspective of one of the web service endpoints in a composition: it combines message exchanges with both internal and external services together with a task model to limit permissible behaviour to that which achieves the endpoint's aims. As this view is limited to one endpoint, only message exchanges that involve that endpoint will be included; the complete message exchange model is thus distributed among the endpoints. Choreography's concern is with the observable behaviour, specifically, the messages exchanged between all participants. Private implementation details of the participants are not included in this view. Therefore, the choreography view will most often not be executable as the orchestration view can be, as its model of the observable behaviour of a set of web services will not capture a complete model of their implementation. These two different perspectives are addressed by different standards in the web services stack, of which a representative example of each is discussed below.

First, we will examine the Business Process Execution Language for Web Services [BPEL4WS], whose name is often abbreviated to BPEL4WS or just BPEL. This standard defines an XML-encoded language for modelling business processes from the perspective of one workflow endpoint, where the basic unit of work is web service messaging. As such, this standard falls neatly into the orchestration category of web service composition standards. Process specifications defined using BPEL can be abstract, where the specified process is limited to the details that define the public role of the modelled endpoint. Consequently, some activities will then be modelled non-deterministically to reflect the hidden nature of their implementation details. Abstract processes are useful when one wishes to model the message exchange protocol without specifying the details of a composition's execution, leaving these to be specified later or alternatively using the model of the process to validate existing implementations. Executable business processes can also be specified, which define the endpoint's business process in its entirety. While an executable business process specified in BPEL may leave certain details unstated, and thus is not sufficient in of itself to be executed by a given BPEL execution engine, the missing details are not at the business process level, and thus the processes themselves will execute and exchange messages with their peers in a consistent fashion, regardless of the execution environment. These two specification styles allow BPEL to benefit from information hiding, both as an implementation technique and as a requirement in the corporate environment.

The authors of the BPEL standard incorporated many features into the language they designed. The following quote from the standard's introduction sheds light on their motivation:

> What are the concepts required to describe business protocols? And what is the relationship of these concepts to those required to describe executable processes? To answer these questions, consider the following:
>
> - Business protocols invariably include data-dependent behaviour. For example, a supply-chain protocol depends on data such as the number of line items in an order, the total value of an order, or a deliver-by deadline. Defining business intent in these cases requires the use of conditional and time-out constructs.
>
> - The ability to specify exceptional conditions and their consequences, including recovery sequences, is at least as important for business protocols as the ability to define the behaviour in the "all goes well" case.
>
> - Long-running interactions include multiple, often nested units of work, each with its own data requirements. Business protocols frequently

require cross-partner coordination of the outcome (success or failure) of units of work at various levels of granularity.

—§ 1: Introduction, [BPEL4WS]

To satisfy their data-driven requirement, the BPEL authors included constructs for the creation, manipulation, and communication of fine-grained XML fragments and also conceptually useful WSDL messages. In BPEL, the core process unit is the *activity*, of which these constructs are subtypes. Added to these were additional activities for time-out, exception signalling, conditional and repeated execution, scoping and exception handling; all necessary additions for process modelling. The standard also defined other constructs to be used to support the definition of these core process units.

Since WSDL messages are XML-based, and thus feature a tree structure, they can be manipulated at any given depth without a complete understanding of the structure of content at greater depths. Unfortunately, business processes will often have to make use of data embedded in the exchanged messages for controlling their execution, for example, examining the data to evaluate a conditional expression. In BPEL, the constructs that support data manipulation may operate with or without the knowledge of the message's structure, allowing for both ease of use and encapsulation where each is appropriate. In the case of a private structure, BPEL provides extensions to WSDL to define and bind *message properties*, which are named fragments of WSDL messages. Once these properties are defined, BPEL processes that reference them will not be affected by structural changes to the WSDL message, so long as the message property bindings in the WSDL file are also changed. For situations where this separation introduces needless complexity, BPEL also allows direct XML fragment access through XPath expressions.

Fundamental to BPEL's data support is the familiar concept of a variable. Variables are typed either as storage for an XML-Schema-conforming XML message fragment, or alternatively for a WSDL message or message property. Variables are given value either through message exchange or through direct manipulation. The BPEL activity for data manipulation is the *assign* activity, which can be used not only to copy data between parts of variables, but additionally from constants and limited XPath expressions to variable parts. In abstract processes, the source of an *assign* may also be opaque, to indicate its absence from the model. Variable values are accessed when evaluating conditional expressions or the duration and deadline expressions used in the BPEL *wait* activity.

Excluding the basic data manipulation facilities detailed above, web service message exchange is the *only* way to modify state in a BPEL process. This is a deliberate decision on the part of the authors; it ensures that the business process remains loosely coupled to the implementation of its constituent steps. There are three activities defined in BPEL to be used in message exchange: *receive*, *reply* and *invoke*. The way these are used depends on the WSDL operation which they implement or with which they interface. (The authors of BPEL only recognize the one-way and request-response operation types defined in WSDL.) For one-way operations, used for asynchronous message exchange, a *receive* alone is sufficient for a BPEL process to implement the operation, while an *invoke* is used to send the message. For request-response operations, used for synchronous message exchange, a *receive* together with a *reply* is necessary to implement the operation, while an *invoke* with additional configuration is used to both send a message and receive a reply. All of these activities store or retrieve their messages from variables.

Rather than having these operations bind directly to WSDL definitions, BPEL introduces partner, partner link and partner link type constructs for both coupling and modelling reasons. Partner link types associate a role with each end of a communicative relationship, as illustrated in Figure 2.2, "BPEL Partner Link Types". The role consists of a WSDL port type that defines all the operations that the web service endpoint filling that role must provide. A partner link type may leave one role unspecified if no operations are necessary, for example if the interaction is a simple message delivery. Partner link types are defined as extensions to WSDL, and then imported into BPEL specifications to create partner links. Partner links associate the named roles with either the modelled web service endpoint or the partner endpoint with which it communicates. Partner links may also be aggregated through the use of partner constructs, to ensure that a single endpoint fills multiple roles. Finally, as it is pointed out in chapter 8 of [ACKM04], BPEL does not define a mechanism by which partner links are associated with web service endpoints at deployment time. However, the *assign* activity may be used to associate an endpoint's address with a partner link at run time, where the address is taken from an XML fragment formatted according to the WS-Addressing standard.

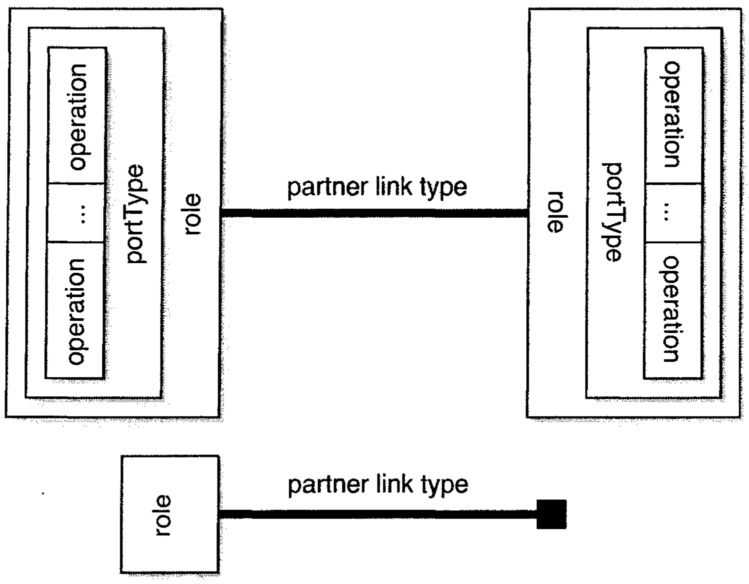


Figure 2.2. BPEL Partner Link Types

In order for a BPEL process to be the recipient of a message that is not part of a synchronous exchange that it has initiated, it must implement some WSDL operation. For executable BPEL processes, receipt of at least one message is made a requirement. The BPEL execution model specifies that the creation of a process instance be in response to an incoming message. This intuitively makes sense, as a stateful business process will likely be triggered by an event, which in this environment would be modelled as a web service invocation. However, since a BPEL process is not limited to implementing one web service operation, some method of associating incoming messages to separate instances of a given process is required. In BPEL, this problem is called "message correlation", and its solution relies on the message property construct explained previously. Essentially, the solution is to compare the values of selected properties of incoming messages to values gathered from previously received messages. The relevant message properties are identified in the message exchange activities where the messages are sent or received, and they are associated with a group of values called a *correlation set*. Once initialized through the receipt or sending of a message, a correlation set contains the values of the message properties that all future incoming messages involved in that correlation set and the current process instance must share. Any given exchange between two partners may involve many correlation sets, each initialized either by messages correlated with a previously initialized set or by the message that started the process' execution.

In addition to the activities described above, BPEL also provides activities to add structure to the process. Conditional execution is provided by a *switch* activity, and the *while* activity provides looping functionality. The *sequence* activity models sequentially executed activities, while *flow* activities may contain any number of activities to be executed simultaneously, optionally with a set of dependencies used to define an acyclic execution graph. The *pick* activity may be used to wait for any number of messages, or optionally a timeout, performing the specified actions once the event occurs. Constructs and activities are also provided for exceptional conditions. Fault handling can be added to either the process or a nested *scope* activity, where action can be specified in the case of WSDL fault messages or *throw* activities. Scopes may also have compensation handlers, which specify actions to take to reverse the effect of the scope's execution. These are generally invoked (through the use of the *compensate* activity in a fault handler) in response to faults that cause a process to fail. These features may be used in concert with a WS-Coordination coordinator to implement transactions.

Because BPEL is the result of a consolidation effort in the web services composition space, it faces few competitors. However, despite the benefits of the orchestration view, such as the possibility of execution, the restriction to a single web service endpoint's point of view makes many verification, distribution and synchronization tasks non-trivial. For this reason, the domain of choreography languages is still active. Many of the languages mentioned by Peltz in his overview [Pel03] are no longer undergoing active development. However, one promising initiative in this space is the Web Services Choreography Description Language, called WS-CDL, a standard that is currently a last-call working draft [WS-CDL] undergoing development at the W3C.

The advantage of the choreography view that WS-CDL adopts is that of consistency. A choreography is defined from a third-party viewpoint, modelling the behaviour of all the participants simultaneously, and in such a model, the participant's behaviour is consistent. The equivalent collection of orchestration views, each modelling the behaviour of one of the participants, may be inconsistent because they are separate models. Indeed, one of the usage scenarios for WS-CDL choreography models that the authors propose is to generate abstract orchestration models, either to be stored using an orchestration-style web service composition language such as BPEL, or to be used to generate skeleton implementations in a traditional general-purpose language. When used in this way, the choreography models would guarantee consistency among the various participant-specific orchestration models, while leaving only the implementation details to the endpoint developers. Choreography models could also be used in system verification tests, where WS-CDL specifications would serve as an authoritative version of an interaction, with which implementations could be compared directly or indirectly through generated abstract orchestration views. Of course, there are other advantages to having a global interaction model, including more accurate modelling of multi-participant concepts such as message relaying and state alignment. Once WS-CDL is complete and tool support becomes available, its specific features should ease the development of composed web service systems. However, since many of its features are useful only during development, many of the deployment issues that affect the existing orchestration solutions will still exist if ever WS-CDL becomes widely adopted.

2.1.2. Web Service Stack Research

There has been much research interest in web services, much as there have been in both industrial and hobbyist realms. Published research on web services is vast and diverse, to match the diversity of the various web service standards and the scope of their application domain. Here, the discussion is limited to the issues surrounding web service composition.

Benatallah *et al* describe a series of patterns for web service composition and execution in [BDFR02]. In that work, service composition is divided into static and dynamic composition. The static composition pattern is meant to address the scenario where the relationships between entities are fixed, and the process used to describe the composition is thus also static. The paper suggests that static compositions be specified at a high level, limiting their scope to control and data flow, and that the compositions themselves should be services accessible in standard ways, enabling inclusion in further compositions. This approach resembles that taken by the web service composition standards discussed above. Moreover, the authors also present two composition execution patterns, centralized and peer-to-peer. In the centralized pattern, the composition is executed by one endpoint, while in the peer-to-peer pattern, the composition occurs between a collection of endpoints that exchange messages not only with the composed services but also amongst themselves when executing the composition. Research that explores web service composition described by these patterns is discussed in Section 2.1.2.1, "Static Service Composition Research". Benatallah *et al*'s dynamic composition pattern addresses the scenario where the composed services are selected automatically from a set of available services. A later paper [BDFRS02] subdivides this pattern into separate "service wrapper" and "service discovery" patterns. The service wrapper pattern allows the components of a dynamic composition to interoperate despite their heterogeneity of data formats and interaction protocols, while the service discovery pattern involves the use of descriptive metadata to select services such that can these be composed together to satisfy the composition's constraints. The area of study known as "Semantic web services" explores the use of dynamic composition, and selected research from this area is discussed in Section 2.1.2.2, "Semantic Service Composition Research".

2.1.2.1. Static Service Composition Research

Assuming a certain complexity of the web services included in a given composed web service, due to their number or their diverse domains, the amount of resources required by the composition's execution will be significant. Distributing this composition across multiple endpoints would likely increase performance and possibly bring scalability and concurrency benefits as well. In order to distribute a composition, some method for decomposing its centralized orchestration model into a set of smaller orchestration models that intercommunicate is necessary. Such a decentralization mechanism is defined in [NK04], which features a thorough analysis of the problem as well as the algorithm. The algorithm operates on a web services orchestration, analysing the data and control dependencies present and ensuring that the semantics of the composition are identical in both the centralized and decentralized versions. This work is part of the larger Symphony project, which also implements and tests the approach [CCMN04]. The implementation decomposes centralized BPEL specifications into smaller BPEL-specified components that compose a subset of the originally composed web services, and adds message exchanges between these components. These decentralized components are then deployed so that they are colocated with the services with which they interact, and perform their portion of processing there. This processing increases the amount of work that is performed at any given location, but the amount of data that is exchanged in the decentralized version of the application is often greatly reduced, as any aggregation or filtering of the data is performed prior to network transmission. Empirical evaluation of these performance claims was a component of both papers, confirming that distributed compositions are beneficial. Additionally, the methods used in the Symphony project eschew the compatibility challenges that manifest themselves when constructing a distributed composition with orchestration models, as they assume a correct orchestration model as input and will not introduce any such errors during decentralization.

Unfortunately, many situations will involve constraints on composite service decentralization such that human intervention will be required. Moreover, in the case of adding interaction between two pre-existing web service nodes such that they become a new decentralized composite application, a complete orchestration model is difficult to construct. In fact, in the case of B2B applications, such a complete orchestration model is undesirable due to the implementation details that would be made public, justifying the need for choreography standards and their associated testing tools. Choreography models, or the abstract orchestration models generated from them, will however need to be compared with orchestration implementations; thankfully, because of BPEL's limited scope, the use of formal methods to aid in verification and validation is often feasible. Furthermore, such techniques are useful in the absence of a choreography model for testing the orchestration models themselves, prompting some research in this area.

For example, [FBS04] describes a system for analysing and proving properties of BPEL specifications. The system uses guarded automata to capture both the flow of control and data, allowing interesting statements made in a temporal logic to be proven by a suitable reasoner. A separate effort uses finite state machines derived from BPEL specifications as a matching tool [MWF05]. Since a match does not indicate protocol compatibility, however, the usefulness of this technique is limited to querying BPEL repositories by example. Another application of formal methods can be found in [FUMK05], which has a much narrower goal: to ensure that deployed BPEL specifications implement the desired collaboration properly. In that work, the focus is on assuring that desired behaviour as captured in message sequence charts (MSCs) is in fact the behaviour that the various per-participant BPEL specifications implement, while previous work [FUKM04] addressed the issue of identifying mismatches between the interaction protocols of each participant. Both works use a translation process that first transforms BPEL and MSC specifications into finite state processes, and then into labelled transition systems (LTSs). These LTSs are then analysed to discover any inconsistencies. In addition to providing a testing environment for validation, the use of the tools developed as a component of these works can prevent incompatible orchestrations from being deployed by ensuring that the orchestration models of all pairwise combinations of the deployed endpoints are compatible.

2.1.2.2. Semantic Service Composition Research

While static composition can be facilitated by the work discussed in the previous section, the task of composition remains the responsibility of human composers. A group of researchers intend to relieve them of this burden by automating the process of composition through the use of semantic description. Succinctly, if the semantics of web services are described in sufficient detail, a web service composition could be constructed automatically, given some semantic description of the composition itself. Semantic description is subdivided in [POSV04] into these categories: data, functional, quality of service, and execution. Data semantics constitute the constraints on the type, value and structure of the data exchanged in web service messages. The functional semantics specify the functional constraints on the inputs, outputs and effects of a web service's operations. The quality-of-service semantics specify the various non-functional qualities of a web service. Finally, the execution semantics specify the process that a given web service implements, and the associated data and control flow. These attributes are those that must be described in order to be able to compose web services dynamically.

Various methods are proposed for enhancing web service description to include the data, functional, quality of service and execution semantics required by dynamic web service composition. The METEOR-S project's MWSAF effort [POSV04] annotates WSDL descriptions using ontologies for data, functional, and quality of service semantics. The described method is agnostic towards the choice of ontology encoding languages, of which there exist several, as long as they have an XML serialization. The semantic data is then used to perform matching between different service descriptions. The OWL-S effort [OWL-S] is another example of a semantic web service description approach, adding ontology definition for execution and functional semantics to the data semantics ontology definition method already provided by traditional OWL. Finally, the WSMO project [FD05] proposes yet another method for semantic description, defining its own ontology language for this purpose.

Once the semantics of a service are described, automatic composition with that service becomes possible. The METEOR-S project implements a form of automatic composition through service matching [AVMM04]. Their method requires that execution semantics be provided in the form of an abstract process specified in BPEL. These abstract processes are annotated with constraints on those services of their partners that are to be bound dynamically, specifying the required semantics for each of them. A pool of semantic descriptions of available services is then provided to an algorithm tasked with finding an appropriate match. This method does not take into account the execution semantics of the composed services, however, leaving this to the human designer of the abstract composed process. A similar matching approach was integrated into a BPEL execution engine in [VAGDL04], where the semantic annotation was performed using OWL and OWL-S. The Astro project, on the other hand, is a focused effort to provide automatic composition using execution semantics alone [PTBM05]. Given a set of requirements for the composite service, together with a collection of abstract process descriptions in BPEL describing the available services, a composed process description (in this case, an executable BPEL specification) is produced. This is done through the use of formal methods, and involves translation between BPEL specifications and state transition systems. These research efforts demonstrate that automatic service matching is possible, but the other component of automatic service composition, service wrapping or adaptation, is also required to prevent minor data format or functional variations between services from impeding composite service construction. The WSMO community has made this a focus of their efforts, naming this procedure “mediation”. Their data mediation process aims to translate between different data representations by constructing a corresponding mapping between the ontologies used by each representation and applying it to the data instances [MC05]. Process mediation is used to hide small differences in web services protocols by introducing a mediation component as an intermediary between services [CM05]. Data mediation implementation has occurred both as part of the WSMO effort [HCMOB05] and outside of it (for example, [MM03]); implementation of process mediation is still forthcoming. However, both types of mediators remain important components of a completely automatic service composition system.

2.2. Agent Communication Research

The previous section reviewed the web services approach to distributed systems. The presented techniques feature process descriptions that structure previously specified message exchanges between services. Different process description methods are proposed for capturing the global and endpoint-specific views. This section explores the work of the agent communication community, which shares an interest in examining message exchange as it occurs in distributed systems of agents.

2.2.1. Agent Communication Overview

The term "agent" is used in many contexts in the research community, so much so that there exists work whose aim is to catalogue and distinguish them [Nwa96]. A subset of the "collaborative agent" group identified in that work, and the particular body of research in which this work is concerned, is that of "agent communication". The agent communication field explores the issues that are associated with distributed problem solving through the use of collaborative software agents, where each has access to different resources and functionality. Typically, these agents are "intelligent", in that they can reason about their operation and their state. In order to direct this reasoning, the belief-desire-intention (BDI) model for agency was adapted from folk-psychology (see for example [RG95], [GPPTW88]). In this model, beliefs encompass the agent's knowledge of its world, or more plainly its state. Desires, in turn, represent a state of the world that the agent wishes to bring about. Finally, intentions represent series of tasks that the agent has selected to satisfy some desire. Bratman's work [BIP88] postulates that intentions are important to limit the amount of replanning done by agents. To facilitate communication between BDI agents, standard agent communication languages (called ACLs) were developed, which could be used by agents to communicate about any given domain. Two such proposals are significant: KQML [LF97] and the ACL of FIPA, the Foundation for Intelligent Physical Agents [FIPAActs]. An ACL message consists of a type label called a *performative*, a message content container, and various fields oriented at message routing and dispatch tasks, such as conversation identifiers, and sender and receiver addresses. A system of BDI agents communicating using an ACL is commonplace in agent communication literature.

However, it was discovered that ACLs, with their rich set of performatives and arbitrary content, were not sufficient to ensure an unambiguous interpretation for any given message. Greaves *et al* called this the "Basic Problem", and defined it as follows:

> Modern ACLs, especially those based on logic, are frequently powerful enough to encompass several different semantically coherent ways to achieve the same communicative goal, and inversely, also powerful enough to achieve several different communicative goals with the same ACL message.
>
> —§ 1: The Roots of Conversation Policies, [GHB00]

In that work, the authors then proceed to propose a solution to the problem, which they call "conversation policies". Conversation policies are meant to specify what messages are allowable in a given inter-agent interaction. Many such policies would exist, each governing a facet of a given conversation: 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. These policies would be declaratively specified, and available to and interpretable by all agents involved in an interaction. Taken together, these conversation policies would specify a normative communication model, which would define for a given agent and its state which messages would be acceptable to that agent. Deciding what should be included in such conversation policies was left as a topic for additional research.

2.2.2. Task-Based Communication

The use of a task environment model to define agent conversations was an idea contributed by the TÆMS project [DL93]. This project, the main attribute of which was a rich and detailed task environment model description language, explored the effect of the structured tasks on the relationships between the agents meant to execute them, particularly when coordinating task execution [DL95]. Later work also explored in greater detail the effects that tasks and conversations have on each other [WBLX00]. This approach causes the conversation to become subordinate to the agent's tasks: the decision to communicate rests entirely on whether the agent, in its current state, requires the communication to achieve its objectives.

Other research in this area has also taken a task-based approach to communication. For example, the COOL language [BF95] provided constructs for defining rules governing state transitions within a conversation to aid in the coordination between agents. Moore's work [Moo00] uses statecharts to model arbitrary task structures, which may include conversation actions. This is also the approach of the RETSINA project [SPVG01], which uses deterministic finite automata (DFA) to define task-level conversation protocols [EPTS01] that are then combined to achieve the goals of the agent [PKPSS99].

Given a structured model of the task to be performed, together with an understanding of how such a model describes agent behaviour, an agent *A* can determine whether sending a particular message is appropriate given its current state. Likewise, if its partner agents also have the task description, and are thus able to infer some fraction of *A*'s internal state based on their own task execution state and the messages that have already been exchanged, they will be able to determine whether *A* is expecting a message from them, along with any restrictions on the type or content of any such message. The task model and its interpretation therefore serves as a conversation policy that can be consulted to determine whether a particular message is expected, or exceptional.

But what is to be done when an exceptional message is received? One proposed method is a centralized exception handling service [KD95], which allows agents to focus on the implementation of their normative behaviour. This centralized solution does not align well with the distributed nature of agent systems, however. An alternate approach is proposed in [NU00], where the agents use small protocol units to define their conversations, dynamically selecting these for execution. Thus, in response to an unexpected message, agents can engage in an error sub-conversation when necessary, returning to the parent conversation upon successful completion. This work presents the dilemma of what to include in such an error conversation. Not surprisingly, there exist performatives for failure-handling messages in the dominant ACLs. In the FIPA ACL, there are *not-understood* and *failure*, while in KQML there are *sorry* and *error*. In [EP05], Elio and Petrinjak distinguish between the receipt of messages that fall outside the normative communication model and other exceptional conditions, proposing that *not-understood* be used exclusively for signalling the former. The authors have also chosen to adopt a model of the tasks distributed across agents as a normative communication model for the agents involved. Having done so, they are thus able to identify the set of conditions under which a message falls outside the communication model, and is thus subject to a *not-understood* reply. Pointing out that such exceptions to the normative communication model indicate that the model is not identical between the involved agents, the authors identify but do not resolve the challenge of how to determine which model is authoritative.

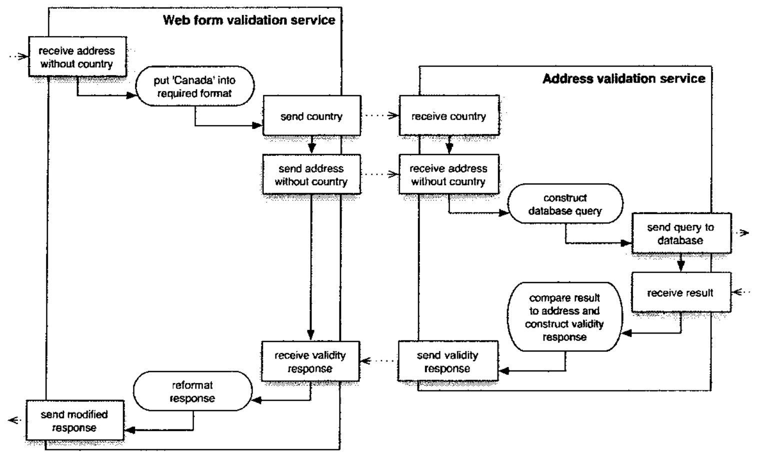
2.3. Problem Scope

To date, we have examined two bodies of literature describing two approaches to the distributed execution of process-oriented tasks. In the web services literature, these tasks are implemented by web services that are aggregations of other services whose composite execution is structured using a process description. Such composition is made possible by the availability of a set of existing web services to compose, each already described and made accessible through standard web service techniques. The compositions themselves are constructed and modelled using a variety of different methods. The construction can be automatic, using semantic descriptions of various aspects of the task and the available services, or the behaviour of each node or web service endpoint involved in the distributed execution of the composition can be specified by developers. Specifications for describing behaviour, constructed either manually or automatically, can take the form of a set of detailed models for each web service endpoint, the strategy of orchestration modelling languages such as BPEL, or might instead model only the externally visible behaviour of the entire set of endpoints from a global viewpoint, the approach taken by choreography languages such as the forthcoming WS-CDL.

The agent communication community literature examines how a set of agents can execute such tasks through the use of inter-agent communication. The nature of the task, combined with the varying abilities of each individual agent defines the content and timing of such communication, given a messaging model and a shared interpretation of the effects of a message on an agent's state. Such a normative communication model can be used to detect messages that are not expected, and a shared policy on such conversation failures can dictate a suitable response. Since deviations from the communication model indicate that the understanding of the task's nature is inconsistent between the agents, such a response could involve the resolution of these inconsistencies to avoid future communication failures.

Before a detailed explanation of the problem that this work is meant to solve, and the elements of the approach of these communities that are used as part of the solution, a unified vocabulary is presented.

2.3.1. On Workflow Models



Web form validation service
receive address without country
put 'Canada' into required format
send country
send address without country
receive validity response
reformat response
send modified response
Address validation service
receive country
receive address without country
construct database query
send query to database
receive result
compare result to address and construct validity response
send validity response

Figure 2.3. Example Workflow

The term *workflow* is given to a process that accomplishes some objective, usually in the context of an organization such as a business, that requires more than a single participant to complete. As a consequence, a workflow typically involves steps that route information among the participants so that the objective may be achieved. Workflows can be conceptualized as *workflow models*, which have often been represented as a state space in which the edges represent actions taken by the workflow's participants. These actions can include both activities performed by a participant in the workflow or alternatively a message exchange between participants. As an example, consider the workflow featured in Figure 2.3, "Example Workflow". The participants in this workflow are the web server (not shown), the web form validation service, the address validation service, and the address information database (also not shown). The objective of this workflow is to validate an address that has been entered into a web form, for example checking that the postal code is feasible given the rest of the address. To model this workflow, one would have to include not only the actions depicted in the diagram, but also those missing: the actions of the web server and web form validation service. The existence of such a workflow model is unlikely, however, for many reasons. First, such a model would increase in size and complexity with the number of participants and the corresponding increase in states. Second, and likely more importantly, workflow models that describe inter-organizational workflows will include descriptions of the internal processes of each organization that is involved. In our example, suppose that the address validation service and the address information database belong to a shipping company, while the web server and the web form validation service belong to a retailer. Control of such a global workflow model would have to be shared between the participant organizations, thus placing external dependencies on any modification of these internal processes, an unacceptable situation for many organizations. Additionally, knowledge of internal processes might be useful to an organization's competitor, and thus disclosure of these processes to external parties is undesirable. For these reasons, subsets of the entire workflow are modelled in practice.

One such subset corresponds to the orchestration view of a web service composition introduced by the web service composition literature. Recall that the orchestration view describes a web service composition from the point of view of a single participant, where the composed web services that interact with that participant are associated with a process model describing when communication with these services can occur and how this communication affects the behaviour of the modelled participant. If the web service composition were to be viewed as a workflow, then any particular web service in the composition is a participant in that workflow, and an orchestration model of that web service includes the necessary work and communication steps found in the subset of the workflow model limited to that participant. This work will refer to such a subset as a *workflow script*. If each participant's behaviour is described by a workflow script, then the details that make up a global workflow model are distributed among this collection of scripts. This piecewise approach to modelling is not limited to the web services arena: the agent system modelling methodology proposed in [KGR96] also suggests that interaction protocols be developed separately for each identified role filled by an agent, and the conversation models of the COOL system [BF95] and the system in [Moo00] (both described above) are participant-specific. Within the web service community, BPEL is used for specifying such workflow subsets, and an executable BPEL specification (together with the companion WSDL specifications it makes reference to) may serve as a declarative description of a workflow script, with a notable addition. As BPEL does not specify a method to bind the partners it defines to web service endpoints, or define any method for imposing constraints on the partners other than their required interfaces, this information must be provided separately. With this addition, a BPEL specification becomes a workflow script specification that completely describes a web service endpoint's operation. Two workflow scripts from our running example may be found in Figure 2.4, "BPEL Specifications as Workflow Scripts". It is important to note that when a workflow model is distributed into many workflow scripts, each of which is then captured in various specifications, it is subject to uncoordinated piecewise modification. In other words, if a developer were to change the BPEL specification that described the workflow script of one of the participants, the other workflow scripts will not be affected as a result. When participant-specific actions are changed, this is a non-issue; however, if the modelled message exchanges between two participants are changed in only one participant's workflow script, then communication failure will likely result.

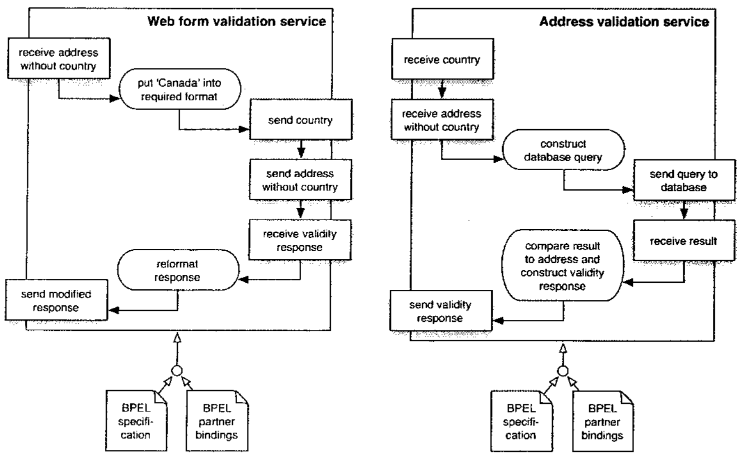

Web form validation service
receive address without country
put 'Canada' into required format
send country
send address without country
receive validity response
reformat response
send modified response
BPEL specification
BPEL partner bindings
Address validation service
receive country
receive address without country
construct database query
send query to database
receive result
compare result to address and construct validity response
send validity response
BPEL specification
BPEL partner bindings

Figure 2.4. BPEL Specifications as Workflow Scripts

Communication failures occur when a particular workflow participant receives a message that conflicts in some way with its expectations. For example, the message's type, content and/or sender could be different from what was expected, or perhaps no message was expected by the recipient given its current state. The participant's response to such a message will result in an exception, or perhaps undefined behaviour. Such failures are the result of a pair of inconsistent workflow scripts, where the message exchanges specified in one script do not correspond to those specified in another. In the agent communication literature, message exchanges that occur between agents in support of a goal are called a *conversation.* Because messages are meant to induce the recipient to perform a certain action or send another message that will advance the workflow towards completion, such agent conversations have been studied in detail, and are viewed by some as a consequence of distributed abilities and the nature of the task to be performed. In the web services world, the choreography view is also interested in this aspect of workflow, and is seen as a way to prevent these communication failures by centralizing a model of these message exchanges. A choreography does not include the details of each participant's actions, but only their conversations with each other. A diagram of the features of the example workflow that would be retained in a choreography is presented in Figure 2.5, "Example Conversation". This restriction makes choreographies a reasonably sized subset of a global workflow model, but also renders them unexecutable. We call these subsets of a global workflow model *conversation models.* As a conversation model is unexecutable, its primary utility is in verification and validation. Such models could be used to verify that the messages exchanged by the participants in a workflow are compatible by comparing the conversation model of that workflow to the collection of workflow scripts that define the behaviour of its participants before deployment. In the agent communication literature, such conversation models are used to determine the acceptability or understandability of messages received by an agent at run-time.

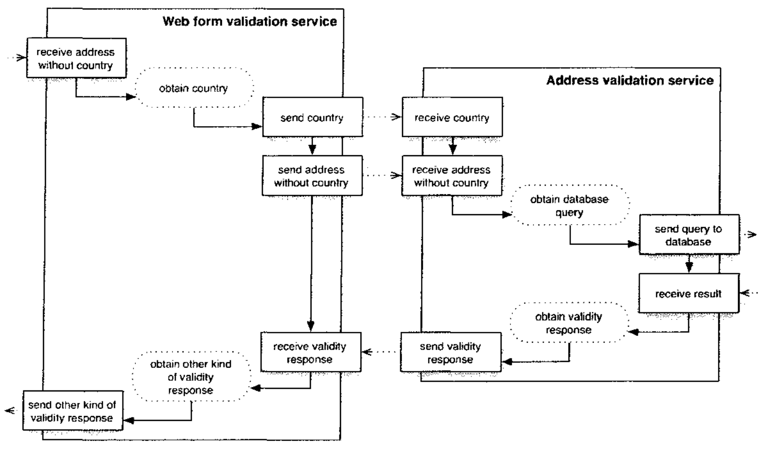

Web form validation service
receive address without country
obtain country
send country
send address without country
receive validity response
obtain other kind of validity response
send other kind of validity response
Address validation service
receive country
receive address without country
obtain database query
send query to database
receive result
obtain validity response
send validity response

Figure 2.5. Example Conversation

These conversation model uses would be facilitated by the extraction of the fragment of a conversation model that describes the conversation between two participants in a workflow, which we call a *conversation script*. If the workflow scripts of both participants agree with the conversation script and are therefore compatible, it is notable that both of these contains all the information that can be found in the conversation script. Thus, a conversation script can be viewed as an intersection between two compatible workflow scripts each defining the behaviour of one end of a conversation. Conversation scripts can therefore be derived from workflow scripts, allowing the agent communication community's run-time conversation model validation techniques to be used without requiring additional models. Pre-deployment validation requires a specification of a conversation model, but until the choreography specification language area stabilizes, specifications of the conversation scripts that make up the conversation model could address this need. Fortunately, conversation scripts may be specified using the same technology as workflow scripts, using these to encode the conversation subset of one of the two participants. Since conversation scripts are missing the details required for execution, these workflow scripts, and the BPEL specifications from which they are derived, will be abstract.

2.3.2. Conversation Script Inconsistencies

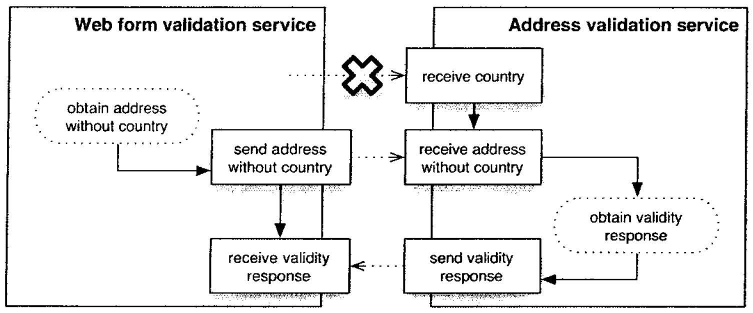

Web form validation service
obtain address without country
send address without country
receive validity response
Address validation service
receive country
receive address without country
obtain validity response
send validity response

Figure 2.6. Inconsistent Conversation Script

Given this vocabulary, the intention of this work as presented in the introductory chapter can be restated: to construct a system that addresses the challenges associated with conversation failures between two workflow participants caused by an inconsistency in the conversation script shared by both. Furthermore, in the environment used in this work, the workflow participants are web service endpoints, and the workflow is a web service composition. Figure 2.6, illustrates one possible example of an inconsistent conversation script using the example workflow. Such inconsistencies result from the independent implementation of the workflow at each endpoint. The most obvious approach to address this issue would be to ensure that such inconsistency does not occur. This could be accomplished by examining the implemented conversation scripts that describe each conversation between two workflow participants, and verifying that each is consistent. These conversation scripts can be extracted from the workflow scripts that describe the implementation. However, obtaining workflow scripts for a workflow participant when it is implemented using a general-purpose programming language such as Java is difficult and time-consuming. Moving the implementation of each service endpoint to an executable declarative specification of a workflow script, such as a BPEL specification, makes compatibility verification quite straightforward. Because of BPEL's restricted scope, however, the implementation of any processing steps would have to remain in a general-purpose language and be refactored into private web services. For any service that features a sufficiently complex interaction protocol, the benefits of moving to BPEL and thus becoming able to avoid communication errors through testing would outweigh the cost of this refactoring.

Since choreography specifications that capture communication models have yet to become popular, tools for the extraction of conversation scripts from workflow scripts do not yet exist. However, it is likely that choreography specifications will also be used to produce abstract workflow scripts describing the conversational behaviour of a participant, so that these may be compared to the workflow script that specifies that participant's behaviour using existing comparison methods such as those proposed in [FUKM04] and discussed above. In the absence of choreography specifications, these abstract workflow scripts must be constructed by hand. Since the abstract workflow script describing a participant's behaviour is simply an inversion of the message exchange protocol specified in the workflow script of its peer, the script's construction should be relatively straightforward. Such abstract scripts would also be useful development aids, as they allow the implementation of a workflow participant to remain private while providing a means to distribute the message exchange expectations of that participant. Thus, the typical strategy will likely resemble that described in [PTBM05] (again discussed above), where implementers of a service provide abstract BPEL specifications to their prospective partners, describing only the pairwise interaction between that partner and the service from that partner's point of view. Assuming the existence of such abstract workflow scripts, specified in a suitable language such as BPEL, together with tools for verification and validation, conversation model discrepancies can be eliminated at development time. However, this does not address protocol discrepancies that may occur during deployment.

For example, consider the case where the owner of a web service decides to modify the conversation script that governs its interaction with one of its partners. In developing the new version of the web service, the owner follows all of the practices identified above, publishing a new abstract BPEL process corresponding to the partner's role. Unfortunately, deploying the new service and the corresponding abstract workflow script will not affect the partner's service until it is made aware of the change. At this juncture, the classic trade-off between push and pull update strategies applies. If the deployment step adopts the push strategy, notifying all partners of the change, then the particular web service endpoint implementing each partner must be known (there can be no anonymous partners). The alternative pull strategy requires that the web service endpoints that are potentially partners in the interaction check the public abstract workflow script to see if it has changed. This strategy avoids the need for an exhaustive list of the potential partners to be maintained at the modified service endpoint, but introduces the need for a schedule to determine when to fetch the public abstract workflow script. At one extreme, the potential partner could check the abstract workflow script before every script execution, avoiding being out of date but checking at the highest possible frequency. Ideally, the potential partner would only check the abstract workflow script when it had changed in a way that would affect the next interaction. For example, if the exchange's purpose is to place an order for goods, and the change affects only the extended exchange that occurs for orders with totals above a certain amount, then the new script would not be necessary until a larger order was to be placed. The network costs of the push method rise with the number of potential partners and the frequency of the modifications. Thus, the pull method is less network-intensive when the frequency of public script checks between modifications is smaller than the number of potential partners. When the optimal schedule is used, and potential partners only check the public abstract workflow script after it has changed in a manner that will affect its next interaction, this requires that some number of potential partners must not be affected by any given update, either because their next conversation happens to not be affected or because they will not communicate at all until after the following update. The decision to use a push or pull technique for abstract workflow script distribution will therefore depend on the usage profile of the potential partners of the service, and also the relative difficulty of maintaining a list of the potential partners at the updated endpoint. This work is predicated upon the decision that a pull model is best.

Once the decision has been made to use a pull model for distribution, conversation model discrepancies due to modification will affect interactions only in situations where an abstract workflow script check is not scheduled between a script change and a script execution. Unfortunately, as the frequency of checks in the schedule increases, so does its distance from the optimal schedule, and so long as there is sufficient time for a modification to be deployed between the scheduled check and an execution of the script, conversation script inconsistencies remain possible. This scheduling problem is an opportunity to apply the techniques developed in the agent communication community and reviewed above. If web service endpoints are cast as agents, then the workflow scripts defined by the BPEL process specifications may be used to determine when the communication scripts are disjoint by examining the messages exchanged at run-time, as described in the work of Elio and Petrinjak [EP05]. In response to such detection, the authoritative communication model found in the public abstract workflow script can be retrieved and the discrepancy eliminated. This scheme mimics the effect the optimal schedule, retrieving the script only if it has changed in a manner that affects this particular execution. The associated cost of this scheme, which is not analysed further in this work, is the cost of reversing the aborted script.

However, once the discrepancy has been eliminated, all that has been gained is the knowledge that the potential partner's current implementation must be modified to conform to the conversation script. Of course, the resolution of the discrepancy could consist of informing a human developer that the implementation requires adjustment; thankfully, better solutions are certainly possible. The semantic web service research described above is in large part predicated on the semantic annotation of web service descriptions, which can be time consuming. Additionally, automatic composition is known to be a computationally hard problem (see for example [PR90]). Nevertheless, the idea of constructing a web composition dynamically is appealing, and could be applied to this problem. Initially, we have a collection of BPEL processes, some abstract, some executable, and the desired end product is an amalgamation of these processes into an executable composed process. When using abstract BPEL processes, the missing details are always values for variables, as these are the result of either missing transformation or message exchange steps. Thus, the composition step becomes the task of matching the types of the missing values to the types of the values available after the execution of a non-abstract process, and connecting the two such that the value is transferred between them during execution. This matching method is quite shallow when compared to the semantic web services work, and should therefore be replaced when frameworks and tools for more sophisticated approaches are more widely available. Despite its simplicity, if this matching method were used to dynamically compose a workflow script from a given set of other workflow scripts defined using BPEL specifications such that all abstract workflow script dependencies were satisfied, then a modified version of one of the abstract workflow scripts used could be integrated into the composed workflow script automatically, assuming that it introduced no new dependencies that were not satisfied by an available executable workflow script. An example of a composed workflow script is provided in Figure 2.7, "Example Composition", which illustrates how the web form validation service's workflow script could be composed from an abstract workflow script describing its behaviour together with the abstract workflow script provided by the address verification service. This means-end analysis composition method, augmented with recursion as prescribed by [BDFR02], is sufficient for small data sets; a similar algorithm is used for data mediator selection in [MM03], for example. Since a web service endpoint is likely to use only a small subset of available services, this method is sufficiently powerful for the dynamic integration of modified abstract workflow scripts needed by this work.

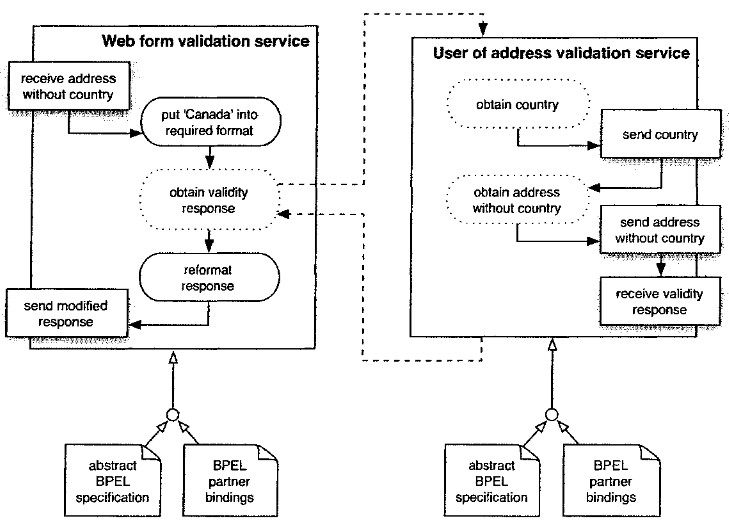

Web form validation service
receive address without country
put 'Canada' into required format
obtain validity response
reformat response
send modified response
User of address validation service
obtain country
send country
obtain address without country
send address without country
receive validity response
abstract BPEL specification
BPEL partner bindings
abstract BPEL specification
BPEL partner bindings

Figure 2.7. Example Composition

These strategies for managing the deployment of workflow scripts defined by BPEL processes will have to be implemented, but what implementation component should be modified? Monitoring the conversations and execution state of a BPEL process in order to detect conversation mismatches is feasible, but only through the emulation of a significant portion of a BPEL execution engine. Likewise, the use of automatically composed BPEL processes becomes a simpler task if the composition is performed within the engine. The use of custom or modified BPEL execution engines is not uncommon in the research arena. For example, this technique has been applied in the addition of aspects to BPEL [CF04]. Further, we have already mentioned Verma *et al*'s use of a modified BPEL execution engine in [VAGDL04]. Limiting modifications to the execution engine allow some level of interoperability with existing BPEL work, such as the aforementioned formal method-based testing tools.

To summarize, conversation script inconsistencies can be avoided at development-time through the use of BPEL and available verification tools, and at deployment-time through the use of conversation model inconsistency detection methods adapted from the agent communication community. Additionally, updated abstract workflow scripts, as defined in abstract BPEL processes, could be dynamically integrated into a partner's execution through a basic application of dynamic service composition techniques. Since the development-time communication model inconsistencies are already resolvable as a result of previous efforts, this work's contribution is in its proposed solution to the deployment-time challenges. The solution is implemented in the Workflow Reconfiguration through Agent- and BPEL-Based Intercommunication Technology (WRABBIT) system, a custom BPEL execution engine which detects and reports conversation model inconsistencies, retrieves the authoritative version of the conversation script as it is stored in an abstract workflow script, and dynamically composes it into the workflow script to be executed by the engine. A detailed description of this system's implementation and capabilities is the subject of Chapter 3, *Solution Implementation.*

2.4. Related Work

With the scope of this work thus established, it can now be compared to other research work. The implemented system builds on the work of the web service and agent communities, which have many affinities. In an attempt to define the unique characteristics of agent-based software systems, Petrie [Pet01] identifies several key components of agent systems: an adherence to a shared agent model, communication through the use of text-based messages with a standard format, and a shared language with which to communicate about domain-specific concerns. Web services are built on easily parsable text-based messaging, and semantic efforts aim to enable interoperability between different arrangements of domain-specific data, leaving only the shared agent model as a distinguishing trait. Blake's work on the WARP environment [Bla02] uses agents as workflow middleware, taking advantage of the agents' reflective abilities to coordinate the execution of component-based services. While these services were not implemented as web services as part of that work, Blake supposed that the approach would work for actual web services as well. The later COACHES environment [Bla04] confirmed this, taking advantage of the interoperability of heterogeneous agents, in this case agents having access to different web services, to execute workflow processes. These works, however, do not address the problem of misaligned agent workflow models that is the subject of this work.

Blake's research featured a global workflow model shared by all agents in the implemented system, but this work argues above that such an approach is unrealistic because it does not permit information hiding. Interestingly, Buhler and Vidal have also used agents for the execution of workflows [BV04], but their work adopts an approach similar to that of this work, where per-agent workflow model segments are specified declaratively using BPEL. Their work postulates that adaptive agents could modify the workflow to add flexibility, however, they do not discuss how these modifications would affect the agent's interaction protocols and if so, how these could be repaired. Further, their reported implementation efforts do not yet support such agent-initiated modifications [BV05]. Nevertheless, these efforts confirm that using BPEL as a model for normative agent interaction is feasible.

Chapter 3. Solution Implementation

As stated previously, the problem that this work is meant to address is that of inconsistent communication models involving web service endpoints. Recall from the discussion in Section 2.3, "Problem Scope", that this problem's solution consists of the following components:

1. The use of a restricted-scope language such as BPEL to implement each endpoint's behaviour, allowing the use of automatic verification/validation tools to check for inconsistent communication models at development time.

2. The public distribution of a set of abstract workflow scripts, each describing a partner's interaction with the endpoint from the partner's point of view without revealing the endpoint's private implementation or dictating the partner's implementation beyond the scope of the conversation script.

3. A distribution method for these abstract workflow scripts that fits with the needs of the system's designers and has the lowest network costs.

4. A mechanism that permits the automatic integration of new versions of these abstract workflow scripts into the implementation of the partner's web service endpoint.

Previous work has provided the verification and validation tools required to address the first component. The construction of abstract workflow scripts is not a difficult task, and in the future will likely be facilitated through the use of choreography models and associated tools. This work addresses the last two components by supplying a workflow script execution engine with features inspired by work done in the agent communication and semantic web communities. This engine monitors its communications with other such engines to determine if its conversation scripts are being adhered to. If a conversation script is violated, an authoritative version of the abstract workflow script from which it is derived is obtained, integrated into the engine's model, and the process is restarted.

This chapter provides more details on the design and operation of the constructed system, called the Workflow Reconfiguration through Agent- and BPEL-Based Intercommunication Technology system, or WRABBIT system for short. The main components of the WRABBIT system are described in Section 3.1, "Agent Architecture", along with the rationale for the system's behaviour. Operational details, such as algorithms and run-time attributes are described in Section 3.2, "Agent Operation".

3.1. Agent Architecture

The workflow script execution engines in WRABBIT are called agents, after the agents of the agent communication community. They are similar to other implementations of agents: WRABBIT agents communicate using an ACL and structure their execution using intentions. However, they do not feature any reasoners or general-purpose planners, which are often found in such systems. Their main function is the execution of workflow scripts, and as these include as a primary component a BPEL specification, WRABBIT agents are a form of a BPEL execution engine. As discussed in the previous chapter, the verification and validation benefits of BPEL's declarative model are most valuable when the complexity of a service is in its composition logic: the target area that BPEL specifications are meant to model. As such, BPEL specifications and workflow scripts that include them will have to interact not only with other similarly-modelled web service compositions, but also web services who interact with other systems or perform complex computations; services not easily modelled in BPEL. WRABBIT agents are only meant to replace web service endpoints whose complex communication protocols with many other services are easily modelled in BPEL. The agents therefore use their agent features only with partners modelled using workflow scripts, and operate with other web services using traditional means. Thus, the WRABBIT system can be described as distributed middleware for executing web service compositions, with the distinguishing feature being its failure recovery policy. The agents themselves also operate as web services, using an XML serialization of the ACL messages when communicating with other WRABBIT agents. A diagrammatic overview of the architecture of a WRABBIT agent can be found in Figure 3.1, "WRABBIT Agent Architecture".

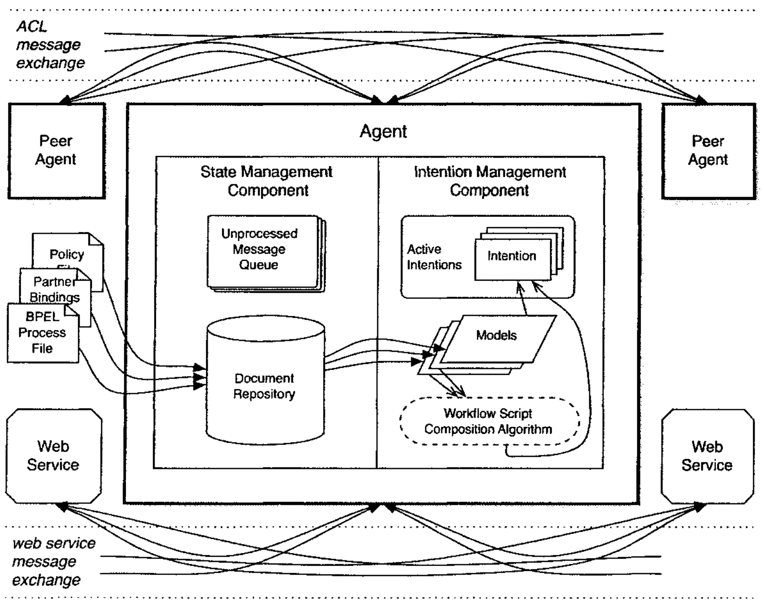


Figure 3.1. WRABBIT Agent Architecture

There are two key components of a WRABBIT agent: its state management component and its intention management component. The state management component manages the information that is shared by the entire agent, such as unprocessed messages, and declaratively specified documents from which models that guide the agent's execution may be constructed. This execution, on the other hand, is the responsibility of the intention management component, which manages the intentions that prescribe the various behaviours that a WRABBIT agent may exhibit at any given time. Intentions each manage their own private state, which may include the aforementioned models derived from the specifications located in the agent's document repository, specifically workflow scripts and error resolution policies. A detailed discussion of intentions follows in Section 3.1.1, "Intentions", while details of various aspects of their operation may be found in Section 3.2, "Agent Operation".

Agents in the WRABBIT system use an ACL to communicate with their peers. The ACL used is not a full implementation of either the FIPA ACL or KQML standards, but rather a custom ACL that incorporates a small set of features from each of these. The agents use web services infrastructure to exchange their ACL messages, allowing this mechanism to benefit from the infrastructure already developed in this field. While WRABBIT agents may execute workflow script with a mix of agent and traditional web service endpoints as partners, their interactions with non-WRABBIT partners are not similarly enhanced, as these service endpoints cannot modify their behaviour in response to failures as can WRABBIT agents. Such interactions thus use standard web service mechanisms, which are not further explained in this work. More details on the use of ACL messaging in WRABBIT along with how it integrates with traditional web service communication features can be found in Section 3.1.2, "Messaging".

Finally, the stated goal of this work is automatic recovery from conversation failures at run-time. With this in mind, a taxonomy of these failures together with an analysis of their causes can be found in Section 3.1.3, "Failure". The section continues by including a discussion of the effect that such failures have on the execution of workflow scripts, how these failures are signalled to other agents, and the strategy that this work adopts for automatic recovery from these failures.

3.1.1. Intentions

A WRABBIT agent's execution is accomplished through the use of intentions of various types. As mentioned previously, these resemble the intentions used in the agent communication community, where an intention is a plan of action which an agent has committed to effectuate, and that when followed, will advance the agent towards the satisfaction of its desires. This commitment to a plan has the useful feature of limiting the amount of costly replanning required in an agent system; replanning becomes necessary only when the difference between the planning costs and the value of the benefit gained from switching plans is positive [BIP88]. Agents in the WRABBIT system do not have a model for representing their desires; however, some intentions in this system do use an algorithm to compose an executable workflow script from a set of other workflow scripts, which can be seen as a very rudimentary form of planning. Replanning, therefore, also exists in WRABBIT agents in situations when workflow script compositions are reconstructed in response to a failure that relates to the original composed workflow script; in this case the value of replanning is obvious and well worth the effort. Beyond these superficial similarities, however, WRABBIT's intentions are quite different from those of intelligent agents; most importantly, their behaviour is fixed, a property of the type of each intention. This conceptualization of intentions maps closely to that of the object-oriented classes that are used to implement them.

The most commonly used intention type in a typical WRABBIT agent is the *Workflow Script Execution* intention. As its name implies, instances of intentions of this type are charged with the execution of an executable or composed workflow script. The detection of conversation failures is therefore also part of the behaviour of this intention, along with the required signalling of these failures. When this happens, the intention enters a failed state, indicating that it did not achieve its purpose of a successful workflow script execution, but recovery from this failure is not the responsibility of this type of intention. Rather, the state of the intention will be used by its parent intention to determine whether recovery is required and what recovery action is appropriate. When the intention fails, it captures details of the particular symptoms exhibited by its failure, to aid in any recovery effort.

This ability of a WRABBIT intention to launch and monitor the execution of other intentions of different types is key to the operation of a *Get Typed Value* intention. Each instance of this intention is configured with a WSDL message type, which is then used as input to the workflow composition algorithm. The algorithm will either select an executable workflow script that, when executed, will have produced a value of the required type, or will construct a composite workflow script with that same property. This workflow script, thus obtained, is used by the intention to create a *Workflow Script Execution* intention, which is added to the set of active intentions. Having created this sub-intention, the intention moves into a waiting state, until the sub-intention is no longer active. If the sub-intention completes successfully, then the stated goal of the *Get Typed Value* intention has been achieved. Otherwise, the intention examines the failure properties of its sub-intention to determine if any recovery is required. Having performed any necessary recovery, the intention will then try again, asking the workflow composition algorithm once again to provide a workflow script that will provide the needed values. If at any point the algorithm is unable to construct such a plan, the intention fails.

While *Get Typed Value* intentions are useful for directing the agent to execute some workflow scripts in order to satisfy the particular demands of a fickle controller, as would be the situation with human-guided execution, it makes sense to also have an intention to respond to other machines when these use pre-arranged conversation protocols. This is the need addressed by the *Script Execution Spawning* intention. It follows the BPEL model, which specifies that a BPEL process is instantiated in response to an incoming message. However, the *Script Execution Spawning* intention extends this model from executable workflow scripts to abstract workflow scripts as well. In both cases, the intention waits for the message that starts the conversation to arrive. Once this event occurs, in the case of an abstract workflow script, the intention calls the workflow script composition algorithm to obtain an executable composed workflow script. The final workflow script is used to create a *Workflow Script Execution* intention, which as before is added to the set of active intentions. As with *Get Typed Value* intentions, the intention will monitor the execution of the sub-intention, waiting for it to become inactive. However, the intention will continue to spawn new sub-intentions in response to new messages. If one of its sub-intentions fails, the intention examines the details of the failure of its sub-intention to determine if any recovery is required, and if so, to perform it. During recovery, the intention will defer processing of any received messages, resuming its normal processing behaviour only after recovery is complete.

In addition to these three core intentions, there exist a number of supporting intentions as well. Because the failure recovery behaviour is identical in both *Get Typed Value* intentions and *Script Execution Spawning* intentions, it has been extracted into its own intention type, *Conversation Failure Resolution.* Instances of this type of intention are created by an instance of either of the two intention types that require its services, who wait until it signals its completion. Another support service in WRABBIT agents is message handling, which is performed by an intention of the type *Message Dispatch.* Assigning the responsibility of message routing to a single intention removes the contention on the message queue that would result if each intention had to check for its own messages. The agent creates this intention itself during its initialization.

3.1.2. Messaging

The use of ACLs is widespread in the agent communication community. An ACL specification defines a message as having a type, a set of standard message fields used for controlling message dispatch and correlation, and a storage area for domain-specific content. Both FIPA's ACL and KQML define a set of standard performatives; nevertheless, agent systems often deviate from these standards (as pointed out in [Pet01]), adding performatives as required by the task, and ignoring those that are irrelevant in their context. This work embraces this plasticity, and uses its own ACL that consists of a small number of performatives: *inform* and *request* are used to exchange information as required by the workflow, while *not-understood* and *sorry* are used to notify agents of failures of different types. The first three of these are modelled on the FIPA ACL, while *sorry* is taken from KQML. The WRABBIT ACL also features conversation control fields taken from FIPA's ACL message structure [FIPAStruct]. In addition to the *performative* field, and fields for identifying the sender and receiver of the message (for which the WRABBIT system uses URIs), the *conversation-id*, *protocol*, *reply-with* and *in-reply-to* fields are used. The system also adopts an ACL's exclusive use of asynchronous message exchange.

Web service message exchange as defined by WSDL, on the other hand, allows for synchronous as well as asynchronous message exchanges. Because WRABBIT agents are primarily used for the execution of workflow scripts based on BPEL processes, which themselves take advantage of both types of exchanges offered by WSDL, this disparity will have to be addressed. First, however, consider an asynchronously delivered WSDL message and its contents. The purpose of such a message is to convey information from the sender to the receiver. Nothing in the WSDL specification indicates that these one-way message exchanges should be organized into some protocol; rather, this is done at the BPEL, or conversation script level. With request-response or synchronous message exchanges, the WSDL specification requires that a message be sent in reply to the received message. This introduces a different protocol layer, underneath any BPEL-specified ordering on exchanges. The authors of the WSDL standard justify the existence of these message exchange patterns in this way:

> Although request/response or solicit/response can be modelled abstractly using two one-way messages, it is useful to model these as primitive operation types because:

- They are very common.
- The sequence can be correlated without having to introduce more complex flow information.
- Some endpoints can only receive messages if they are the result of a synchronous request response.
- A simple flow can algorithmically be derived from these primitives at the point when flow definition is desired.

—§ 2.4: Port Types, [WSDL]

The last reason listed hints at the solution this work adopts: a mapping between synchronous request-response exchanges and an equivalent series of ACL messages. As part of this mapping, the workflow scripts that the agents maintain will also need to be modified as they are constructed from their corresponding BPEL files to reflect these changes, and additional correlation information will have to be introduced. However, as a result of the introduction of this mapping, the message exchange protocols will be represented at a single level, the ACL level.

To construct this mapping, two ACL performatives were chosen: *inform* and *request*. The *inform* performative is traditionally used when an agent wishes to convey information to another, while *request* is generally used when an agent wishes that another agent would perform some action. Table 3.1, "Mapping of BPEL activities to ACL message protocols", demonstrates how these have been used in the mapping. Because the purpose of a web service message is to convey information, *inform* is featured in all the ACL protocols. Additionally, a *request* message is featured in the protocol whenever synchronous messaging is used, to capture the explicit demand for a reply featured in any WSDL request-response message exchange.

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 3.1. Mapping of BPEL activities to ACL message protocols

Having established this mapping, some method of identifying these messages as related by their membership in a WSDL operation is necessary. This is accomplished through the ACL message properties *reply-with* and *in-reply-to*, adapted from FIPA's ACL. For synchronous exchanges, a token uniquely identifying a particular exchange for a given originating agent is constructed from the identifier of the agent's *Workflow Script Execution* intention and a generated identifier unique within that intention. This token is then placed in the *reply-with* field of both the *inform* and *request* messages sent to the other agent. When the receiving agent sends its *inform* message in reply, it uses that same token, garnered from the previous messages, as the value for the *in-reply-to* field of that message. These two techniques are not sufficient, however, to allow web service messages to be exchanged using an ACL. Agents exchange ACL messages between each other, while web service messages are targeted at web service operations, of which many might be provided by a given agent. For this reason, the *inform* and *request* ACL messages used to transport web service messages feature an operation identifier in the message's content that uniquely identifies a WSDL operation by incorporating the service, port, and operation identifiers that are present in WSDL. In the case of *inform*, the content area is divided to include the contents of the web service message alongside the operation identifier. With these elements in place, the WRABBIT system's ACL acts as a transport mechanism for WSDL messages exchanged with other agents, taking the place of SOAP, for example. The introduction of this new layer allows the agents to enhance the interaction protocol defined by the workflow scripts they are executing with additional messages used to communicate with each other about these script executions.

The remaining two performatives, *not-understood* and *sorry*, are in fact used by agents to communicate about workflow script execution. Specifically, these performatives are used to signal failures of different types that the agents have encountered during workflow execution: *not-understood* is used for conversation failures, where the previous message in the conversation is unexpected, or in other words, when the receiving agent cannot understand the reason for the message it has received; *sorry* is used in cases where the messages in the conversation were expected, but the agent is nevertheless unable to complete its task or meet its obligations. The occasions when these message types are used and their contents are discussed in later sections; however, there remains the question of how these messages are associated with the ongoing workflow script conversation that they describe. This is where the ACL message field *conversation-id* is used. Whenever a WRABBIT agent sends an ACL message, the *conversation-id* field is populated with the identifier of the intention that caused the message to be sent. In the case of *inform* and *request* messages, when received by the other agent, this value, together with the sender's agent identifier taken from the *sender* field, can be associated with the workflow script conversation of which the message is a component. This having been done, subsequent *not-understood* and *sorry* messages featuring these same conversation identifiers can be associated with these executing workflow scripts, as long as they originated from the same intention. The saved *conversation-id* values may also be included in the content of *not-understood* and *sorry* messages, allowing such messages to be associated with workflow script executions in cases where this message is the first reciprocal message in an exchange. More details on this technique may be found in Section 3.2, "Agent Operation".

Since WRABBIT agents run as web services themselves, they must communicate with each other through XML-encoded messages. A straightforward XML serialization is used on the ACL messages so that they can be exchanged using web service techniques. An example is shown in Example 3.1, “Example ACL Message in XML”. Additionally, WRABBIT agents may communicate with traditional web services. However, since these services are not WRABBIT agents, these conversations will not benefit from the agent’s recovery abilities. Also, since WRABBIT agents do not expose their implemented web service endpoints other than through the ACL communication mechanism, these traditional web services cannot require an exposed service of their clients. For example, services requiring a callback are not supported by the current implementation.

```
<?xml version="1.0" encoding="utf-8"?>
<ACLMessage conversationID="ProcessSpawningIntention.122;WorkflowScriptExecutionIntention.147"
            performative="inform"
            protocol="http://…/wrabbit/examples/ProvideStudentRecords/script"
            receiverAgentNamespace="http://…/wrabbit/examples/InstructorAgent"
            senderAgentNamespace="http://…/wrabbit/examples/DepartmentAgent">
  <content>
    <ACLMessageContentEnvelope>
      <messageContent>
        <messageContent>
          <messageContentEntries>
            <messageContentEntry>
              <contentNode>
                <node>request id 189663</node>
              </contentNode>
              <messagePart>
                <messagePart name="requestID"
                             namespace="http://…/wrabbit/examples/StudentRecords/wsdl"
                             type="string" />
              </messagePart>
            </messageContentEntry>
            <messageContentEntry>
              <contentNode>
                <node>this is a student record</node>
              </contentNode>
              <messagePart>
                <messagePart name="studentRecords"
                             namespace="http://…/wrabbit/examples/StudentRecords/wsdl"
                             type="string" />
              </messagePart>
            </messageContentEntry>
          </messageContentEntries>
        </messageContent>
      </messageContent>
      <operationIdentifier>
        <operationIdentifier operationName="receiveRequestedRecords"
                             portTypeName="recordAcceptorPT"
                             portTypeNamespace="http://…/wrabbit/examples/StudentRecords/wsdl"
                             wsdlDocumentNamespace="http://…/wrabbit/examples/StudentRecords/wsdl" />
      </operationIdentifier>
    </ACLMessageContentEnvelope>
  </content>
</ACLMessage>
```

Example 3.1. Example ACL Message in XML

3.1.3. Failure

Since the execution of workflow scripts features prominently in the behaviour that WRABBIT agents exhibit, failures that are related to this activity are an important aspect of this system, especially considering that the stated goal of this work is recovery from these failures. The approach to failure recovery implemented in the WRABBIT system features the following components:

1. A taxonomy of failure types, each with a specific set of symptoms by which they might be identified.

2. A cause or reason that precipitated the occurrence of a particular failure type.

3. A repair action that can be taken to fix the agent's models such that the failure does not re-occur, barring further changes to the agent's world.

4. A policy that can be used to select from the set of possible repair actions that could be taken to correct any given failure.

Failure recovery in the WRABBIT system thus proceeds as follows: failure symptoms occur, allowing a failure of a particular type to be identified. For this failure, some number of repair actions are available, each of which addresses all the possible reasons for that particular failure type and its symptoms. Note that if multiple reasons exist for any failure type, then either the exact reason is not determinable from the symptoms, or the failure type requires division into subtypes with greater specificity. However, such refinement of the failure taxonomy is only useful if repair actions can take advantage of this new information to specialize their behaviour for better results. With this set of possible resolution actions in hand, agents then use a policy to determine which to perform.

The taxonomy of failures will be addressed first. This work suggests that there are three high-level failure categories:

Conversation failure	A conversation failure's symptom is the receipt of a message that is not accounted for in the conversation model, given the receiving agent's current state.
Capability failure	A capability failure's symptom is the inability to begin the execution of a task that was either previously performable or of indeterminate performability.
Execution failure	An execution failure's symptom is the inability to continue the execution of a task that had begun previously.

Note that application failures are not included in this taxonomy, because they are not failures from the agents' point of view. For example, messages used to signal application failures are modelled in the workflow scripts, as is their exchange and processing. While all the failures listed above can likely be further analysed and decomposed, capability failures and execution failures are not the focus of this work, and no further decomposition of these failure types is required. Most crucial to this work is the conversation failure, which can be further subdivided as follows:

- Uninterpretable message content: the agent cannot interpret the content of the received message. For example, agent A might send separate first and last names to agent B, who is expecting a full name instead.
- Unexpected message: the agent is not expecting the received message, given its current state
    - Out-of-order message: the agent can route the received message to an ongoing conversation, but given the conversation's state, the conversation model does not

admit the message. For example, agent A might send two messages in sequence to agent B, while agent B expects agent A to wait for its response to agent A's initial message prior to sending the second one.

    - Operation not provided: the agent is not familiar with the operation identifier used in the content of the message. For example, agent A might send a message to agent B as part of a WSDL operation that agent B has removed from the conversation.

    - Unknown conversation: the agent can identify that the received message is acceptable to a possible conversation, but does not belong to any current conversation and is not a designated conversation-starting message. For example, agent A might send a message to agent B that agent A believes to be the initial message in the conversation, but that agent B believes to be the second message in the conversation.

    - Unknown workflow script: the agent cannot route the received message to an ongoing or possible conversation, and does not recognize the identified workflow script. For example, agent A might send a message to agent B as part of a conversation that should occur with agent C instead.

- Operation type mismatch: the received message is part of a synchronous WSDL operation, and an asynchronous message is expected for the identified operation, or *vice-versa*. For example, agent A might send a message to agent B as part of a WSDL request-response operation, which when complete would oblige agent B to respond. However, agent B would believe that that the identified WSDL operation was in fact a one-way operation, and should not require a response.

- Illegal partner: the received message's type and content conform to the expectations of the agent, but the sender of the message does not. For example, agent A could send a message to agent B, who expects that particular message only from agents C or D.

This completes the failure type taxonomy that is used in the WRABBIT system. In order to address these failures, their causes must first be identified.

This work has maintained that all conversation failures are caused by mismatches between the workflow scripts of the conversing agents, which cause inconsistencies in their conversation scripts. In the case of an "illegal partner" failure, the particular mismatch between the workflow scripts occurs in the partner bindings section, where restrictions on the particular agents that may serve as one of the modelled partners are defined. The mismatch could stem from an evolution in the roles of the partners: some of the responsibilities might have been shifted from one partner to another. Another possibility is that the restrictions for a given partner might have been relaxed or tightened, allowing more or fewer agents to act as that partner, respectively. The cause of an "operation type mismatch" failure is the modification of the operation type of a WSDL operation present in a workflow script. For "unexpected message" failures, the mismatch is also in the BPEL and WSDL specifications, where operations have been added or removed, or where message exchanges have been added, removed, reordered, or replaced with other message exchanges. The particular sub-failure that is detected depends on the type of modification and its location. The "operation not provided" failure, for example, occurs when the receiving agent's workflow script has modified its message exchanges such that all use of one particular operation has been eliminated, or alternatively when the sending agent's script is altered such that a new operation is introduced. The "out-of-order message" and the "out-of-conversation message" failures occur when message exchange alterations do not affect the set of operations (only changing the number of times they are used), when these alterations eliminate an operation in the sender's script, or when these alterations introduce an operation in the receiver's script. The distinction between the two failures merely indicates whether the message exchange designated as the conversation starter is affected by the change. In the case where it has been changed, then "out-of-conversation message" failures will occur, while "out-of-order message" failures will occur only after the successful exchange of the message that the receiver's model considers the conversation-starting message. An "unknown workflow script" failure occurs only when one of the two conversing agents is unaware of the existence of the workflow script in question. Finally, the "uninterpretable message content" failure occurs when the schema specifying constraints for the payload of the message differs between the sender and the receiver.

The causes for conversation errors having thus been identified, possible repair actions for correcting them are now explored. Some form of data mediation, as it exists in the semantic web services community, is required to address "uninterpretable message content" failures beyond the deactivation of the workflow script, as the content of messages will depend on the script's own interpretation of the data. Therefore, this work does not address this type of failure. However, it does propose a repair action that handles "illegal partner", "operation type mismatch" and "unexpected message" failures. First, the occurrence of the failure is signalled to the agent that sent the message, so that both agents are aware of the failure. This is accomplished through the use of a *not-understood* message, whose contents include the failure type and the message that prompted the failure. To correct the mismatch, the agents must come to have compatible workflow scripts. In the environment that is described in this work, in addition to the main workflow script, a developer also creates an abstract workflow script describing a particular partner's behaviour for each of the main workflow script's partners. All the workflow scripts created by this developer are assumed to be compatible, so to avoid workflow script mismatches, these need only to be distributed. This work relies on a shared, declarative policy to identify an authoritative source for any workflow script. The authority policy is indexed using the workflow script's identifier and the type of failure used, with a default authority for cases where the failure type is not important. The repair action uses this authority policy to determine which agent is the authority for the workflow script and the failure type in question. Once an authority is identified, both agents obtain the documents that make up the workflow script they are replacing from this authority, and construct a new workflow script from them. In the environment described above, one of the agents will be the authority for the workflow script; however, the flexibility of this policy allows for the distribution of workflow script authority among different agents. For example, while one agent's developer might ensure that the BPEL specifications are compatible, another agent's developer might decide who should be able to access the other agent's service, and construct the partner bindings file appropriately. The latter agent would be made an authority for the workflow script in case of partner failure, while the former would be the authority for other types of failure. While this repair action may be used to address all of the different kinds of "unexpected message" failures, in the case of an "unknown workflow script" failure, the receiving agent that has no knowledge of the workflow script will likely also not know what agent is the authority on the policy. As this failure occurs only in situations where the agents have not been properly configured, this work does not propose a method to enhance the repair action to deal with this failure. Thus, only one repair action is required to deal with most of the subtypes of conversation failure.

Causes and repair actions also exist for the other two high-level failure types, execution failures and capability failures. Execution failures occur in the WRABBIT system when an executing workflow script detects a failure of another type. The recovery methods proposed in this work do not address the repair of executing instances, but only attempt to prevent the failure from occurring in the future. Thus, the occurrence of any failure during execution of workflow scripts also results in the abnormal halt of the execution, which in turn constitutes an execution failure. As with conversation failures, the occurrence of this failure must be signalled to the partner agents that have exchanged messages that belong to the executing workflow script, as any further messages will no longer be acceptable. The *sorry* message is used for this purpose, with the content of the message indicating the type of the failure and its cause. Since other types of failures entail execution failures, these execution failure signal messages are not sent to the partner with which an existing failure occurred and has already been signalled. When an execution failure signal is received by an agent and associated with an ongoing workflow script execution, it causes that execution to suffer an execution failure of its own. Thus, an execution failure will cascade until all executing scripts in the workflow have been halted.

Capability failures in WRABBIT occur when *Script Execution Spawning* intentions or *Get Typed Value* intentions are unable to obtain a composed workflow script using the workflow script composition algorithm. In the case of *Script Execution Spawning* intentions, this failure needs to be signalled so that the agent that sent the initial message does not pursue the execution of its workflow script. A *sorry* message is again used for this purpose. Upon receipt of a message signalling the occurrence of a capability failure, an executing workflow script will halt, and additionally the workflow script will be removed from the pool of available workflow scripts, as it involves an interaction that will not succeed.

3.2. Agent Operation

The WRABBIT system is implemented using the Java programming language, and as such requires a JVM for its operation. It is built using several Java frameworks: WSDL4J provides WSDL parsing, Jaxen provides XPath processing and Apache Axis provides web service access and provision support. Further details on the operation of WRABBIT agents are discussed below.

3.2.1. Agent Run Loop

A WRABBIT agent's behaviour is controlled by various typed intentions, as explained previously. Although an agent queues received messages in a separate thread of execution, an agent's collection of intentions is processed sequentially. At any given time, an intention may be waiting for some event before further execution is possible. For example, the intention may need to process a message, and the current message to be processed is not the desired message. Intentions often delegate work to sub-intentions, and wait for these to complete their work: another example of an event. Therefore, intentions have the ability to list a set of events that are necessary before execution can proceed. Further, the agent needs to know the state of an intention so that it can remove inactive intentions from the collection. For this reason, intentions provide state information to the agent upon request. The execution of an agent thus proceeds as shown in Procedure 3.1, "Agent Run Loop". The agent begins the execution of this run loop immediately after completing its initialization process.

Procedure 3.1. Agent Run Loop

```
1.  Repeat
    a.  For each active intention $i$
        i.   Determine whether $i$ is waiting for an event
        ii.  If $i$ is not waiting for an event then
             A.  Perform $i$'s tasks
```

```
            B.  If i is no longer active, then
                I.  Remove i from the collection of active intentions
        b.  If all intentions were waiting for events, then
            i.  Wait for any of the specified events to occur
```

3.2.2. Agent Initialization

When an agent is first activated, it is provided with an identifier (in the form of a URI) along with a more human-readable name. This identifier is used wherever a token uniquely identifying the agent is required, for example in ACL messages or declarative documents that specify authoritative script determination policy. The identifier does not provide information on the network location of the identified agent, however. It is therefore important that no two agent instances be initialized with the same identifier. An agent will create three intentions as part of its initialization process, before starting the agent run loop. First, it creates a *Workflow Script Execution* intention that executes a workflow script providing documents used to construct workflow scripts in response to requests for these. This behaviour is necessary to ensure that the proposed repair action is able to obtain the authoritative version of a workflow script, where the authority may be any agent. Second, a *Message Dispatch* intention is created to dispatch the messages the agent receives. Finally, a *Configuration* intention is created. This intention, which was not discussed previously, is responsible for processing the messages used for configuration of agents.

3.2.3. Agent Configuration

Agents in the WRABBIT system, once initialized, are configured using special configuration messages. These ACL messages use a special performative, *configure*, to communicate configuration information to agents. The use of a special, non-standard performative differentiates this detail of the particular system implemented as part of this work from the relevant ACL communication on which the proposed solution depends. Three different aspects of an agent may be configured using this method. First, agents can be made aware of the network locations of other agents by adding or replacing associations between agent identifiers and network locations. Second, agents can be provided with documents from which they may construct workflow scripts. Third, an agent can be made to add a new intention to its collection of active intentions. Two different types of intentions may be created in this way: *Get Typed Value* intentions, and *Script Execution Spawning* intentions. For *Get Typed Value* intentions, the configuration must include identifiers for the WSDL message type and the WSDL file in which the message type definition can be found, while for *Script Execution Spawning* intentions, the identifier of the workflow script to be executed is necessary. These configuration messages are processed inside the normal agent run loop through the use of a *Configuration* intention.

The information that is included in a workflow script is found in several separate documents. The process description aspect of a workflow script, for example, is found in a BPEL specification, which in turn may refer to any number of WSDL specifications. The partner bindings file used to restrict the set of agents that are allowed to fulfill a given partner relationship is also necessary for the construction of a workflow script. Further aspects of a workflow script, explained in later sections, are included in dependency and linkage files. To manage this collection of files, the WRABBIT system uses a workflow script description file that features a script identifier and includes the identifiers of the documents that contain the details of the workflow script. Upon receipt of a workflow script description document, the agent configuration intention will attempt to construct a workflow script model using the set of available documents, and then add the resulting workflow script to the set available to other intentions. This behaviour creates a convention that the workflow script description document is sent to the agent last during configuration.

3.2.4. Workflow Script Composition

A WRABBIT agent's ability to create a composed workflow script from a set of workflow scripts plays an important role in the system. Without this capability, new versions of abstract workflow scripts describing the agent's behaviour, once obtained from a suitable authority, would require developer effort to integrate into the agent's operation. The algorithm takes as input the set of workflow scripts known to the agent, and some description of the composed workflow script. This description is alternatively a WSDL message type that the composed workflow script must produce, or an abstract workflow script that should be included in the composed workflow script, with its information value needs satisfied. The algorithm proceeds in a recursive manner, connecting outstanding needs to workflow scripts that supply the required values until no outstanding needs exist. A workflow script is seen as supplying an information value of a particular type if it has a variable of that type included in its BPEL specification, which is assumed to have a value after a successful execution. An abstract workflow script is seen as having a need for a typed information value wherever an opaque assignment is made to a typed variable in the script's BPEL specification. However, this simplistic modelling of dependencies is often insufficient. For example, the contents of certain variables should perhaps not be available beyond the scope of the particular BPEL process. Thus, a script may optionally include dependency information in a separate file.

The dependency file describes the typed information values that will be available after the workflow script has completed its execution, and those that it requires during its execution. In the dependency model, these are called *supplies* and *needs*, respectively. Supplies simply identify a BPEL process variable by its name to indicate that the value of that variable will be available. Needs identify the abstract assignments that they describe by using the name of the BPEL *assign* activity that features the opaque assignment. The task of the algorithm thus becomes finding supplies to satisfy needs. In certain circumstances, however, more information is required about an information value need, specifically, which information values are available at the time that the need must be satisfied. This allows an abstract workflow script to supply the needed value while simultaneously requiring a value of the same script it is supplying. This situation resembles a function invocation, where values are provided by the invoking scope to the function's scope in return for a required value. To address this requirement, need descriptions also include supply descriptions that indicate which variable values are available at the point in the workflow script's execution where the need must be satisfied. The description file thus consists of a set of need descriptions, each featuring a set of supply descriptions, to describe the workflow script's information value requirements, and also a set of supply descriptions to describe the information values available after its successful execution.

As mentioned above, the workflow script composition algorithm operates in a recursive fashion, with each recursion triggered by the addition of an abstract workflow script to the provisional composition. The algorithm's operation is summarized in Procedure 3.2, "Workflow Script Composition Algorithm". The algorithm takes as input a set of needs to satisfy, a set of workflow scripts that may be used to satisfy the needs, and a partial composition of workflow scripts. In the case where the algorithm is invoked using a need for a typed information value, the partial composition will be empty, while in the case where an abstract workflow script is used, the partial composition will contain that script and its needs will be used as the *needsToSatisfy* parameter. This abstract workflow script may contain a *receive* activity in its BPEL specification that is marked as starting the execution of the workflow script; however, the collection of scripts must not include any other such script. This restriction prevents composed workflow scripts from having multiple entry points, which would make them significantly different from uncomposed workflow scripts.

Procedure 3.2. Workflow Script Composition Algorithm

```
composeWorkflowScript( needsToSatisfy, availableScripts, partialComposition )
1.  For each need n in needsToSatisfy
    a.  If n can be satisfied by a workflow script s in partialComposition
        i.   Connect s's supply and n in partialComposition
    b.  else if n can be satisfied by a non-abstract workflow script s in
        availableWorkflowScripts
        i.   Add s to partialComposition
        ii.  Connect s's supply and n in partialComposition
    c.  else
        i.   For each abstract workflow script s in availableWorkflowScripts that
             satisfies n
             A.  Copy partialComposition to provisionalComposition
             B.  Add s to provisionalComposition
             C.  Connect s's supply and n in provisionalComposition
             D.  Invoke composeWorkflowScript( s's needs,
                 availableWorkflowScripts - s, provisionalComposition )
        E.  If the result is not **failure**
            I.  Replace *partialComposition* with the result
            II. Exit loop
    ii. If $n$ is not yet satisfied
        A.  Return **failure**
```

Notice that the algorithm prefers to satisfy needs first by using workflow scripts already in the provisional plan (Step 1.a), then using non-abstract workflow scripts not in the plan (Step 1.b), and finally using abstract workflow scripts not in the plan (Step 1.c.i). Preferring executable to abstract workflow scripts avoids growth of the composed workflow script, as does preferring the workflow scripts already in the plan. These preferences are merely heuristics, however, and are not guaranteed to produce the optimal composition. While the addition of non-abstract workflow scripts (Step 1.b) is straightforward, the other two steps require additional explanation. Step 1.c.i, the addition of an abstract workflow script, iterates over the set of abstract workflow scripts that satisfy the current need, while the other two addition steps simply select an arbitrary workflow script from the available set. This iteration is required in the case of abstract workflow scripts not yet in a plan as these may have needs that are not satisfiable by any available workflow script. Thus, each abstract workflow script must be tested for satisfiability until a satisfiable script is found, or until no more scripts are available. This test for satisfiability is simply a recursive invocation: if the algorithm fails to satisfy the selected abstract workflow script's needs, then that script is not satisfiable. Otherwise, the returned partially composed workflow script, which includes the selected abstract script along with the scripts required to satisfy its needs, is used as the basis for further composition. Step 1.a, the use of workflow scripts that are already present in the partially composed script, seems straightforward, but the introduction of unexecutable circular dependencies must be avoided.

In the WRABBIT system, the components of composed workflow scripts must be executed such that values are available when they are needed. For example, if a workflow script needs a particular value at some point during its execution, the script that provides that value must complete its execution so that the value becomes available. During the providing workflow script's execution, the needy script execution is paused. If both of these scripts depended on values supplied by the other, then they would deadlock: each paused waiting for the other to finish. However, this simple circular dependency is still executable in the WRABBIT system, provided that at the point that one of the two scripts needs a value, it can provide the value that the other needs, preventing deadlock. Thus, not all circular dependencies are unexecutable. An example of an executable circular dependency is illustrated in Figure 3.2, "WRABBIT Executable Circular Dependency". Note that deadlock is prevented in this case, as Workflow Script A is able to provide value 1 to Workflow Script C at the time that it needs value 2. This allows Workflow Script C to complete its execution, providing value 3 to Workflow Script B, which in turn finishes executing, providing value 2 to Workflow Script A. To ensure no unexecutable circular dependencies are created in a composite workflow script, workflow scripts already present in the composition are only allowed to satisfy a need *n* if all of the following conditions hold:

- The workflow script offers the information value required by *n*
- The workflow script is not the owner of *n*
- The workflow script is in the need ancestry of the script that owns *n* and the required value is accessible to *n*.

Some elaboration is required for the last point. A workflow script s_1 is a member of the need ancestry of another workflow script s_2 if a need of s_1 consumes values supplied by either s_2, or a third workflow script s_3, where s_3 is in the need ancestry of s_2. A script in the need ancestry of another script can be associated with a numerical degree indicating the number of association edges that separate them. Need ancestry can alternatively be expressed as a set of logical statements:

$$\forall s_1, s_2(\ \text{needAncestor}(s_1, s_2, 1) \Leftarrow \exists n(\ \text{need}(n, s_1) \wedge \text{satisfiedBy}(n, s_2)\ )\ )$$

$$\forall s_1, s_2, d{\geq}1(\ \text{needAncestor}(s_1, s_2, d) \Leftarrow \exists n, s_3(\ \text{need}(n, s_1) \wedge \text{satisfiedBy}(n, s_3) \wedge \text{needAncestor}(s_1, s_2, d\text{-}1)\ )\ )$$

Using the running example, suppose Workflow Script C's need for value 1 was the only one not yet satisfied. The need ancestry of Workflow Script C thus includes Workflow Script B (because Workflow Script C supplies Workflow Script B with value 3) and also Workflow Script A (because Workflow Script B supplies value 2 to Workflow Script A and is in the need ancestry of Workflow Script C). The accessibility of a supplied information value relies upon need ancestry: a value supplied by a script s_1 is accessible to a need n, belonging to a workflow script s, if all the needs of s_1 satisfied by s or need ancestors of s offer that value, and given that s_1 is a need ancestor of s of degree d, all other ancestors of s of degree d also satisfy this first condition. This too can be expressed as a logical statement:

$$
\begin{aligned}
\forall v, n_1(\text{ accessible}(v, n_1) \Leftarrow\ & \exists s_1, s_2, d(\text{ need}(n_1, s_1) \wedge \text{offered}(v, s_2) \wedge \\
& \text{needAncestor}(s_2, s_1, d) \wedge \\
& \forall s_d, n(\text{ needAncestor}(s_d, s_1, d) \wedge \text{need}(n, s_d) \wedge \\
& \text{offered}(v, n) \Leftarrow \exists s_a, x(\text{ satisfiedBy}(n, s_a) \wedge (\ s_a = s_d \\
& \vee \text{ needAncestor}(s_a, s_d, x)\ )\ )\ )\ )\ )
\end{aligned}
$$

Workflow Script C may therefore access value 1 because it is provided by Workflow Script A's need for 2, the only need supplied by a member of the need ancestry of Workflow Script C (in this case, Workflow Script B). While this set of conditions ensures that composite workflows do not have unexecutable circular dependencies and are therefore correct, the composition algorithm is not necessarily complete, in that there may exist certain valid compositions of workflow scripts that are not admitted by this algorithm. An improved solution would likely involve the application of algorithms used in the semantic web services community or more generally in the artificial intelligence literature on planning. However, the focus of this work is on communication failure recovery and not planning, and thus this algorithm is sufficient.

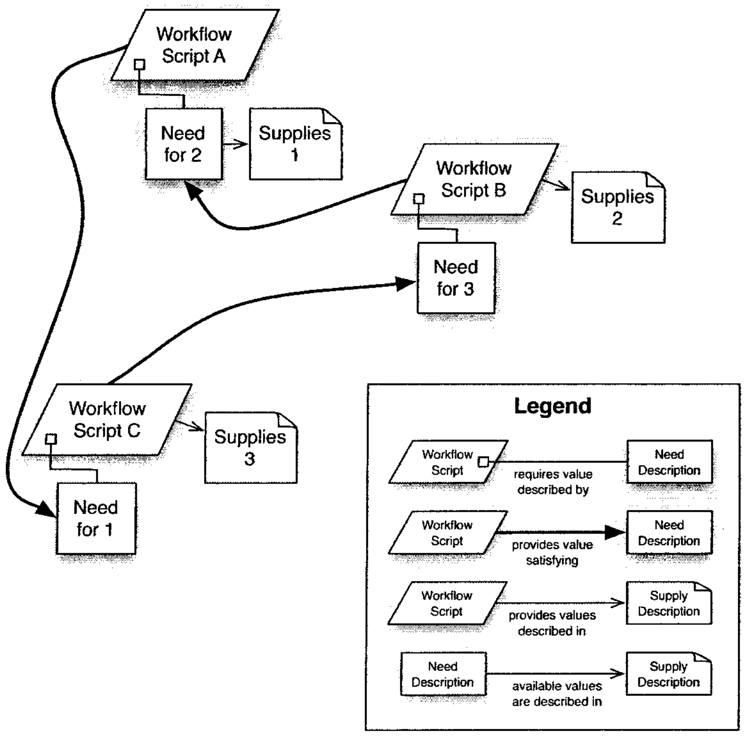


Figure 3.2. WRABBIT Executable Circular Dependency

3.2.5. Workflow Script Execution

Workflow script execution is performed in WRABBIT agents by *Workflow Script Execution* intentions, which are created by other intentions for various reasons. This intention is responsible for tracking and advancing the state of the process specified in BPEL, while monitoring the messages exchanged for conversation failures. When created, *Workflow Script Execution* intentions are provided with either an executable workflow script or a composed workflow script. In the case of a composed workflow script, the intention begins by executing the primary script. When it encounters an opaque *assign* activity in the BPEL process description of the workflow script it is executing, the intention then examines the composition to determine which workflow script must be executed to supply this need. If the supplier script has not yet been executed, the intention begins its execution, pausing the execution of the needy script. Otherwise, and also upon the completion of a workflow script, the value is transferred between the execution state of the supplier script and that of the needy script. The intention is complete when the execution of the primary workflow script has completed.

Currently, the *Workflow Script Execution* intention supports the execution of only a small fraction of the process activities described in BPEL. The supported activities are *receive*, *reply*, *invoke*, *assign*, and *sequence*. This limitation could be eliminated, given sufficient time; however, the current implementation is sufficient for the evaluation of the system as it is performed in this work.

3.2.6. Message Routing and Correlation

When a WRABBIT agent receives an ACL message, it must be routed to an intention that is managing the conversation to which it belongs. Once this is done, the conversation script that governs that conversation, which is derived from the workflow script that is known to the intention, may be used to determine if the message is acceptable. This routing is performed by a *Message Dispatch* intention that is created by the agent during initialization. For any given message, there are three possibilities:

1. The message belongs to an ongoing conversation governed by a conversation script.
2. The message starts a new conversation governed by a conversation script.
3. The message signals a failure.

The first two cases are important to the initiation or pursuit of the execution of workflow scripts and to the detection of conversation failures, while the last case is important to ensure that failure recovery actions are performed. Messages that fall into the first category are handled by *Workflow Script Execution* intentions, which manage the conversations associated with the workflow scripts they are executing. The creation of new *Workflow Script Execution* intentions in response to new messages is performed by a *Script Execution Spawning* intention, and as such, messages falling into the second category are handled by these. Messages signalling failure may trigger recovery actions in a variety of intentions, and these are discussed in the following section.

Several methods may be used to associate a message with an ongoing conversation. For *inform* messages, which carry the payload of the web service message exchange that they replace, BPEL correlation mechanisms are the obvious choice. Upon the receipt of such a message, the operation identifier is mapped to the BPEL activity that receives it, and the relevant parts of the message as located by BPEL message properties are compared to the values in the correlation set. If the data matches, then the message belongs to the conversation, and is acceptable. Once deemed acceptable, further data may be taken from the message and used to initialize values in other correlation sets, which shall be used as the basis of comparison for future messages. However, the correlation data must be initialized prior to its use in correlation, allowing situations where the association of a particular message with the conversation is unknown until other messages have been processed.

It is important to note that BPEL correlation is not required in the case of WSDL request-response operations, so another mechanism must be used for *request* messages sent out as part of the mini-protocol used to emulate such operations in the WRABBIT ACL and the *inform* messages that constitute replies to these requests. In order to associate these with a conversation, the *reply-with* and *in-reply-to* ACL fields are used. Thus, to correlate *request* messages, the receiver of a request-reply exchange stores the value of the *reply-with* field used in the initial inform message, which is associated with the conversation using standard BPEL correlation, and accepts the *request* message if its *reply-with* value matches. The initiator of a request-reply exchange also stores the value used to initialize the *reply-with* field, and uses it to correlate the *inform* message sent in reply.

Finally, in the case where the conversation scripts are not compatible between partners due to the introduction of a new message exchange on the sender side, the operation identifier used in the message is not known to the receiver's *Workflow Script Execution* intention. In this situation, it is no longer possible to associate the message with a conversation with lower-level BPEL or WSDL mechanisms. The message is associated with an ongoing conversation through the use of the *protocol* and *conversation-id* ACL fields. Specifically, if a message's *protocol* field identifies the workflow script of a partner defined in a workflow script currently being executed, and the *conversation-id* matches that used during a previous exchange with this same partner, the message can be associated with the conversation linked with the executing script. This requires that a workflow script include the identifiers of the workflow scripts that its partners are executing, information that is provided in linkage files included in a script's definition.

Intentions of type *Workflow Script Execution* implement all the methods described above for associating a message with a conversation. However, because these intentions also execute composite workflow scripts, these methods must be applied to the set of component workflow scripts, which at any given point alternatively will have finished executing, will be executing, or will not have started execution. The algorithm used by a *Workflow Script Execution* intention is described in Procedure 3.3, "Ongoing Conversation Message Association". Note that this algorithm is only a segment of the complete algorithm, which includes cases for failure signalling messages that are discussed separately in Section 3.2.7, "Failure Detection and Recovery". Other than this exception, this algorithm does include all messages that can be matched to a particular conversation, but as Step 2.a makes clear, messages that cause new conversations to begin are not handled by this type of intention.

Procedure 3.3. Ongoing Conversation Message Association

```
doesMessageBelongToConversation( message )
1.  If message has an in-reply-to field
    a.  If message's in-reply-to field matches our stored value
        i.  return true
    b.  Otherwise
        i.  return false
2.  If the operation identifier in message is referred to in a component workflow script
    a.  If message's operation is used to start executions
        i.  return false
    b.  If message's operation is referred to in a component workflow script that is
        currently being executed
        i.  If message matches stored data (BPEL correlation data or reply-by field
            value)
            A.  return true
        ii. If message does not match stored data
            A.  return false
        iii. If no data is stored with which to match against message
             A. return undeterminable
     c. If message's operation is referred to in a component workflow script that has
        not yet been executed
        i. return undeterminable
     d. If message's operation is referred to in a component workflow script that has
        completed its execution
        i. If message matches stored data (BPEL correlation data or reply-by field
           value)
           A. return true
        ii. If message does not match stored data
           A. return false
3. If the workflow script identifier in message's protocol field corresponds with a
   completed or executing workflow script
     a. If message's conversation-id field matches any known to the execution state of
        the identified script
        i. return true
     b. Otherwise
        i. return false
```

In fact, this type of message is the purview of *Script Execution Spawning* intentions. Here, the task is to associate a message with a conversation defined by a workflow script that has not yet begun execution. This task is greatly simplified if the message is known not to belong to any ongoing conversations, so this assumption is made by the intention. Under normal circumstances, the only message that begins the execution of a workflow script is the one received by a BPEL *receive* activity that is tagged as the execution starting point. However, in the case of disagreements between the workflow scripts of the agents, other messages may also be attempting to serve this purpose. For example, if the message exchanges are rearranged such that the message the receiver agent's script identifies as the one that starts a conversation corresponds to the message that the sender agent's script sees as the second or third message exchanged, then the receiver agent will receive one or two messages before eventually starting the conversation. In order for failure recovery to occur, the receiving agent must identify these messages as outside the conversation model for that script. In the case of messages received whose operation identifiers are present in the receiver's workflow script, but rejected by all ongoing conversations, *Script Execution Spawning* intentions can easily determine their association with that script. Where the operation identifier is not present in the receiver's workflow script, due to deletion on the receiver's side or addition on the sender's side, the ACL *protocol* field is matched to the script's identifier. This yields the algorithm in Procedure 3.4, "New Conversation Message Association", which is used in *Script Execution Spawning* intentions.

Procedure 3.4. New Conversation Message Association

```
doesMessageBelongToConversation( message )
1.  If message is not an inform
    a.  return false
2.  Otherwise if the operation identifier in message is referred to in the workflow script
    to be executed
    a.  return true
3.  Otherwise if the workflow script identifier in message's protocol field corresponds
    to the workflow script to be executed
    a.  return true
```

Since *Workflow Script Execution* intentions and *Script Execution Spawning* intentions are able to determine if a given message is associated to their conversations or workflow scripts, respectively, all that remains is to distribute messages to those intentions that claim an association. This is the task of a *Message Dispatch* intention, whose operation is relatively straightforward. When delivering a message, this intention will first see if the message is associated with a particular active *Workflow Script Execution* intention. If any such intention claims the message, it is dispatched to that intention. In the event where some intentions may be unable to determine the associability of the message, the *Message Dispatch* intention will defer the processing of that message until its next processing session. Otherwise, since the message cannot be associated with an ongoing conversation, the intention will determine if the message initiates a conversation by querying the active *Script Execution Spawning* intention. Again, if any such intention claims the message, it is dispatched to that intention. If no intention claims the message, the *Message Dispatch* intention must respond to it appropriately. In cases where the message is not signalling a failure of some kind, then it falls outside of any ongoing conversation and is therefore an occurrence of conversation failure of type "unknown workflow script". The intention's behaviour for the remaining cases, where the message is signalling failure, is discussed in the following section.

3.2.7. Failure Detection and Recovery

A WRABBIT agent's ability to detect and recover from failure is its defining feature. Recall from the discussion in Section 3.1.3, "Failure" that detection of failures involves the recognition of the failure's symptoms and the identification of its type, while recovery requires some policy that identifies the combination of the available repair actions that then must be applied. The behaviour of WRABBIT agents is found for the most part in that of its intentions; it is no surprise that failure detection and recovery are also performed by intentions.

As was explained in the previous section, the decision to associate an incoming ACL message to a workflow script is accomplished by both *Workflow Script Execution* intentions and *Script Execution Spawning* intentions. These intentions are also responsible for the detection of conversation failures, with the exception of "unknown workflow script" failures, which are detected by a *Message Dispatch* intention when a message cannot be associated with a workflow script. A *Script Execution Spawning* intention can detect two types of conversation failure: "unknown conversation" and "operation not provided". An "unknown conversation" failure occurs when the intention receives a message that features an operation identifier that is referenced in the workflow script to be executed, but is not the operation used to start execution. Such a diagnosis relies on the precondition that the message belongs to no active conversation; a condition shared by the intention. In the similar situation where the message's operation identifier is unknown, but its *protocol* field identifies the workflow script to be executed, the diagnosis is an “operation not provided” failure. A *Workflow Script Execution* intention also detects failures of the type “operation not provided”, but also the types “operation type mismatch“, “illegal partner“ and “out-of-order message”. An “operation not provided” failure is detected in a *Workflow Script Execution* intention when, once again, the message’s operation identifier is unknown, but the message can be associated with the executing or executed workflow script through the use of the message’s *protocol* and *conversation-id* fields. An “operation type mismatch“ failure is detected when an *inform* message is received as the first message of an ACL message exchange sequence implementing a WSDL request-reply operation, but is missing the required *reply-with* field. An “illegal partner“ failure is detected when a message is received from an agent that does not conform to the constraints for the partner relationship as defined in the workflow script that otherwise accepts the message, or alternatively if some other agent has already established itself as that partner for this execution through previously received messages. Finally, an “out-of-order message” is detected when a message arrives before the currently expected message, or after the last expected message has been received.

With conversation failures thus detected, recovery is required. In all cases, WRABBIT agents respond to the detection of a conversation failure by signalling its occurrence, through the use of a *not-understood* ACL message sent to the agent that sent the offending message. The *not-understood* message includes the particular type of failure that was detected, along with a copy of the not-understandable message. Additionally, if the failure was detected in a *Workflow Script Execution* intention, the detection of a conversation failure causes the ongoing workflow execution to fail. The receipt of a *not-understood* message by an agent notifies it of the conversation failure that occurred, allowing it to perform the same recovery actions as its peer, with the exception of *not-understood* signalling. Note that in order for a *Workflow Script Execution* intention to fail in response to a *not-understood* message, so that it may match its peer intention’s state, the message must be routed to the appropriate instance. This is accomplished through the same mechanism as other messages: a *Message Dispatch* intention determines if any of the active *Workflow Script Execution* intentions are associated with the message. The test for association is simple: if the *conversation-id* field of the message that caused the conversation failure is that used by the intention, then the *not-understood* message belongs to that intention, and its execution fails. For both agents, the failure of this intention causes its owner intention (of either *Script Execution Spawning* or *Get Typed Value* type) to begin the workflow script exchange process. This process is also begun within a *Script Execution Spawning* intention, after it has detected and signalled a conversation failure, and within a *Message Dispatch* intention, if it was unable to find a suitable intention to which to deliver a *not-understood* message.

While the points at which the workflow script exchange process is begun are well-defined by the agent's implementation, the process also requires an authoritative source for a given workflow script and failure type, and this part of the recovery policy is stored in declarative policy files. Currently, each workflow script file includes the identifier of the policy file that identifies its authoritative sources. The policy files currently support specific authorities for "illegal partner" failures only, in addition to a default authority for all other failure types. The workflow script exchange repair action thus begins by identifying the agent who is the authoritative source for this workflow script, given the failure type. Once the authority is identified, the set of documents that define a workflow script is requested, one by one, from the authoritative agent. To support this process, all agents, during their initialization phase, create a *Script Execution Spawning* intention that sends documents to agents that request them. The workflow script used by this intention contains a web service invocation to obtain a document that is handled specially by the agent, but is otherwise a normal script, defined by a BPEL specification and other files. Document retrieval is also performed by a standard workflow script. Once all the documents have been obtained, a new workflow script is constructed from them, which is used to replace the previous version. After this repair action is complete, the conversation failure that was originally caused by mismatching workflow scripts is no longer possible, because the workflow scripts of both peers, having been obtained from the same source, are assumed to be compatible.

Besides conversation failures, WRABBIT agents must also handle execution and capability failures. As discussed previously, a capability failure occurs when an intention is unable to construct a composed workflow script using the workflow composition algorithm. When such an intention is of type *Script Execution Spawning*, this failure is signalled to the sender of the message that was to start workflow script execution through the use of a *sorry* ACL message. These *sorry* messages are used to propagate the failure to *Workflow Script Execution* intentions belonging to other agents, so that these would also fail, and appropriate repair actions be taken. Such messages must therefore be associated with a *Workflow Script Execution* intention, and the existing mechanisms suffice for this purpose. In the case of capability failures, the contents of the *sorry* message include the value of the *conversation-id* field of the message meant to begin workflow script execution. Thus, the message is associated with the *Workflow Script Execution* intention identified by that value. The receipt of such a message causes the intention to fail, and its parent intention performs recovery by removing the failed workflow script from its collection, as it depends on functionality that its peer is no longer able to provide.

Both capability and conversation failures thus cause a *Workflow Script Execution* intention to fail. This occurrence is an execution failure, and these are also signalled through the use of *sorry* ACL messages. These *sorry* messages, like those used for capability failures, are meant to cause the execution of workflow scripts to fail in an agent's peers. For any given execution failure, one peer will not have to be notified, either because the execution failure is due to the occurrence of a conversation failure, in which case a *not-understood* signal has already been sent to the peer, or because the execution failure is due to the receipt of some failure signal from a peer, which as such is no longer executing. If any other agent has sent messages to or has received messages from the intention, it must be sent a signal. In cases where the failed intention has only sent messages to a peer, and never received messages, the *sorry* message will not contain the *conversation-id* of the peer's intention, as it is not known. Thus, the conditions by which a *sorry* message is associated with a *Workflow Script Execution* intention must be modified. In the case where the destination intention identifier is not included, the value of the *conversation-id* field of the *sorry* message is matched against those used previously by partners of the workflow script, of which the failed execution must be a member if it had previously sent messages used by that intention. If a *sorry* message cannot be associated with an intention, then the target intention has likely already finished executing, and the message is ignored.

Chapter 4. Solution Evaluation

Previous chapters have explained and situated the problem this work is meant to address, and described the proposed solution and its implementation. To exercise the presented approach and identify issues for further investigation, a series of case studies were designed. The implementation of these case studies also serves as a series of system tests for the WRABBIT software. These case studies are presented below, following a discussion of the domain in which they are situated.

4.1. Case Study Domain

The case studies discussed below are inspired from intuitive scenarios of workflow reconfiguration that might occur within an academic department. Suppose that various people in the department have automated some of their day-to-day activities as web services. For example, the department's administration has a service that provides student transcripts to faculty members, allowing these to include the service in composite workflows such as displaying a student's transcript, or correcting mistyped grades. In the case studies, WRABBIT agents are used to implement such public services using workflow scripts. Any private abilities needed to provide these services are implemented as traditional web services using Java and Axis, which are then used in the workflow scripts provided to the agents. These traditional web services are merely placeholders for the actual services that would access university systems, and therefore dispense meaningless data.

Five WRABBIT agents appear in the cases below: an Instructor Agent and a Teaching Assistant Agent that capture the activities of faculty members and graduate students, respectively, a Department Agent that performs the functions of a member of the department's administration, a Payroll Agent that accomplishes tasks that are handled by a university's payroll group, and finally a Communication Agent that carries out the duties associated with a university's communication office. Each of these agents uses various workflow scripts that involve communication with their peers and also with any web services they require. The focus of the case studies, generally speaking, is on the conversation 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: exactly the situation where the WRABBIT agent's recovery mechanism is triggered.

Each case study described below thus contains a description of the agent's workflow before and after the change, including a rationale for the change that is plausible in the academic environment. An examination of the sequence of message exchanges between the agents follows, in which one or more conversation failures occur, and the WRABBIT system's resolution mechanism is used, followed by a successful re-execution. These execution details were gathered by examining the log files of the WRABBIT agents after each scenario's execution. The first case study, presented in Section 4.2, also includes a subsection that features listings of the exchanged messages along with additional details on how the messages are processed by the WRABBIT system implementation.

4.2. Missing Precondition for Introduced Exchange Case Study

This case study involves two agents, the Department Agent and the Instructor Agent, that experience a conversation failure due to an added message exchange on the Department Agent's side. Additionally, the introduction of this new message exchange adds a precondition in the form of a required information value to the Instructor Agent's workflow.

4.2.1. Case Study Details

The Department Agent offers a service that provides a student's record, including their grades, in response to a request for the record. Thus, the Department Agent publishes the "Student Record Obtainer" workflow script for the transcript-obtaining partner, which is the complement to its own "Student Record Provider" script. When using this script to obtain a student record, the Instructor Agent first sends a request message, and then expects a message containing the record in response, as diagrammed in Figure 4.1, "Workflow Execution before Added Exchange". In response to the request message, the "Student Record Provider" script specifies that the Department Agent use a traditional web service to obtain the transcript (for example, from a database), and enclose it in the message that it returns to the Instructor Agent. The message exchange is composed of two WSDL one-way operations, and as such, the ACL messages exchanged contain no *request* messages. Rather, the interaction uses the 'callback' pattern, and so the Instructor

Agent provides its identifier in the record request message, which the Department Agent then uses to direct its return message.

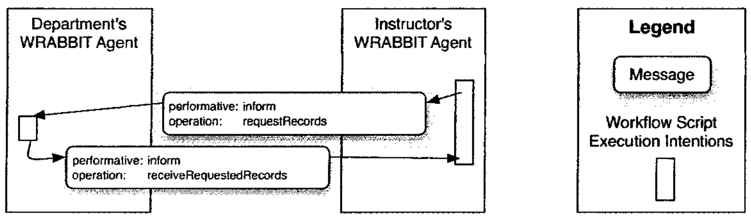

Department's WRABBIT Agent
Instructor's WRABBIT Agent
performative: inform
operation: requestRecords
performative: inform
operation: receiveRequestedRecords
Legend
Message
Workflow Script Execution Intentions

Figure 4.1. Workflow Execution before Added Exchange

Unfortunately, the Department Agent must modify the student record workflow in response to new privacy legislation. To satisfy the legislation's requirements for greater security and non-reputability, the department now requires that all access to student records be tracked. To enforce this, the workflow is changed through the addition of a new initial message that contains an authorization token. These authorization tokens may be obtained from the Department Agent using the "Authorization Token Retrieval" workflow script, and as the tokens have previously been required for other workflows, the Instructor Agent is already aware of this script. The addition of this new message exchange to the workflow has the side-effect of rendering the "Student Record Obtainer" workflow script abstract: it now requires an authorization token prior to execution, a dependency that was not present in the previous version.

4.2.2. Case Study Exchange

In this case study, the Instructor Agent is configured with the unmodified version of the "Student Record Obtainer" script, and aims to locate or compose a workflow script that produces a student record message. Since the workflow script that retrieves the record from the department's agent features the desired message, the system selects the script. The script is not abstract and thus requires no composition prior to execution. The execution of this script initiates the series of message exchanges between the Instructor Agent and the Department Agent depicted in Figure 4.2, "Workflow Execution with Added Exchange". However, as the Department 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 (sequence (a) of Figure 4.2).

The particular conversation failure caused by the Instructor Agent's message and detected by the Department Agent is an "unexpected message" failure, as from the viewpoint of the Department Agent the message no longer initiates a legal conversation. The Department Agent sends a *not-understood* message to the Instructor Agent identifying this failure type, and both processes are terminated. The Instructor Agent, using the content of the *not-understood* message, determines that its "Student Record Obtainer" workflow script does not match the "Student Record Provider" workflow script of the Department Agent. The authority policy dictates that the Department Agent is the authority for this type of failure ("unexpected message") and this particular script ("Student Record Obtainer"). Thus, the Department Agent's failure recovery process takes no further action, while that of the Instructor Agent communicates with the Department Agent to obtain an updated set of the documents that define its workflow script (sequence (b) of Figure 4.2). (Recall that all agents are configured to conduct such conversations with other agents so as to provide any scripts for which they serve as authority.)

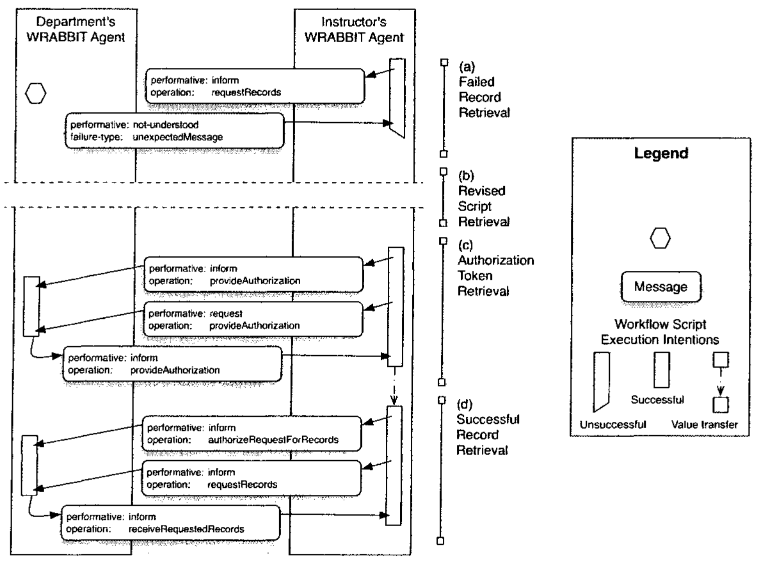

Department's WRABBIT Agent
Instructor's WRABBIT Agent
performative: inform
operation: requestRecords
performative: not-understood
failure-type: unexpectedMessage
(a) Failed Record Retrieval
(b) Revised Script Retrieval
(c) Authorization Token Retrieval
performative: inform
operation: provideAuthorization
performative: request
operation: provideAuthorization
performative: inform
operation: provideAuthorization
(d) Successful Record Retrieval
performative: inform
operation: authorizeRequestForRecords
performative: inform
operation: requestRecords
performative: inform
operation: receiveRequestedRecords
Legend
Message
Workflow Script Execution Intentions
Unsuccessful
Successful
Value transfer

Figure 4.2. Workflow Execution with Added Exchange

Once the updated script has been obtained, the Instructor Agent will again try to locate or compose a workflow script that produces a student record message. Unlike the previous version, however, the "Student Record Obtainer" workflow script is abstract and requires an authorization token prior to execution. Thus, the workflow composition algorithm is used to construct an executable script. When the algorithm selects the "Student Transcript Retrieval" script, it identifies the new precondition of obtaining the authorization token. Since the Instructor Agent was also configured such that it is aware of the "Authorization Token Retrieval" script that retrieves this token from the Department Agent, the algorithm inserts it into the composed workflow script such that it will execute prior to the "Student Record Obtainer" script. This completes the composed script, as it has no further dependencies, and it is executed by executing each of its component scripts as they are needed. The "Authorization Token Retrieval" script executes without issue (sequence (c) of Figure 4.2), as it has not been modified. Also, because the respective scripts for the Department Agent and the Instructor Agent now match, the student record conversation completes successfully (sequence (d) of Figure 4.2).

4.2.3. Thorough Examination of Messages Exchanged

The above description of this case study is appropriate for illustrating the WRABBIT agents' approach to conversation failure recovery and their behaviour in response to this particular workflow modification. However, for the interested reader, an illustration of the operation of the agents' implementation follows in this section, using the messages exchanged during the course of this case study.

Before the WRABBIT agents begin any conversation, they must first be configured both with the appropriate documents with which to construct workflow scripts and also with the intentions that use them. This is done through the use of configuration messages, of which a representative sample is shown in Example 4.1, "Configuration Message". These *configuration* ACL messages, because they are specific to this particular implementation, are not shown in the workflow execution figures of this chapter, such as Figure 4.2, and will not be discussed further.

```
<?xml version="1.0" encoding="utf-8"?>
<ACLMessage conversationID="configuration"❷
            performative="configure"❶
            receiverAgentNamespace="http://…/wrabbit/examples/InstructorAgent"
            senderAgentNamespace="scenario runner"❷>
  <content>
    <ACLMessageContentEnvelope>
      <messageContent>
        <messageContent>
          <messageContentEntries />
        </messageContent>
      </messageContent>
      <operationIdentifier>❸
        <operationIdentifier operationName="activate shutdown"
                             portTypeName="configuration"
                             portTypeNamespace="urn:agentConfig"
                             wsdlDocumentNamespace="urn:agentConfig" />
      </operationIdentifier>
    </ACLMessageContentEnvelope>
  </content>
</ACLMessage>
```

❶ The performative in use is the custom *configuration.*

❷ Since configuration messages are not part of an inter-agent communication, the *conversation-id* and *sender* ACL fields hold meaningless values.

❸ Configuration messages use artificial operation identifiers embedded in their content to pinpoint the desired configuration action. In this particular case, the agent is asked to activate the behaviour of shutting down when no longer active.

Example 4.1. Configuration Message

The Instructor Agent is configured to create a *Get Typed Value* intention that desires a student record. As such, the agent selects the unmodified version of the "Student Record Obtainer" script for execution. The first inter-agent message is sent to the Department Agent from the Instructor Agent. The contents of this message may be found in Example 4.2, "Initial Student Record Request Message". As with any message, the receiving agent's *Message Dispatch* intention will first try to identify an existing conversation as the owner of this message, but as it is the first message sent, no conversation has begun. Thus, there are no *Workflow Script Execution* intentions to claim ownership of the message, and so the dispatch intention proceeds to the next phase, and tries to associate it with a new conversation. Since the message's operation is featured in a possible conversation of the "Student Record Provider" workflow script, the *Script Execution Spawning* intention responsible for starting the execution of that script will accept the message, as prescribed by Step 2 of Procedure 3.4, "New Conversation Message Association". However, from the recipient's point of view, this message does not start a conversation, and its reception causes a conversation failure of type "unknown conversation" to be detected by the *Script Execution Spawning* intention, as described in Section 3.2.7, "Failure Detection and Recovery".

```
<?xml version="1.0" encoding="utf-8"?>
<ACLMessage conversationID="GetMessageContentIntention.120;WorkflowScriptExecutionIntention.121"
            performative="inform"
            protocol="http://…/wrabbit/examples/ObtainStudentRecords/script"
            receiverAgentNamespace="http://…/wrabbit/examples/DepartmentAgent"
            senderAgentNamespace="http://…/wrabbit/examples/InstructorAgent">❶
  <content>
    <ACLMessageContentEnvelope>
      <messageContent>
        <messageContent>
          <messageContentEntries>
            <messageContentEntry>
              <contentNode>
                <node>request id 180046</node>
              </contentNode>
              <messagePart>
                <messagePart name="requestID"
                             namespace="http://…/wrabbit/examples/StudentRecords/wsdl"
                             type="string" />
              </messagePart>
            </messageContentEntry>
            <messageContentEntry>
              <contentNode>
                <node>student record id goes here</node>
              </contentNode>
              <messagePart>
                <messagePart name="studentRecordID"
                             namespace="http://…/wrabbit/examples/StudentRecords/wsdl"
                             type="string" />
              </messagePart>
            </messageContentEntry>
            <messageContentEntry>
              <contentNode>
                <node>http://…/wrabbit/examples/InstructorAgent</node>
              </contentNode>
              <messagePart>
                <messagePart name="endpointReference"
                             namespace="http://…/wrabbit/examples/StudentRecords/wsdl"
                             type="string" />
              </messagePart>
            </messageContentEntry>
          </messageContentEntries>
        </messageContent>
      </messageContent>
      <operationIdentifier>❷
        <operationIdentifier operationName="requestRecords"
                             portTypeName="requestAcceptorPT"
                             portTypeNamespace="http://…/wrabbit/examples/StudentRecords/wsdl"
                             wsdlDocumentNamespace="http://…/wrabbit/examples/StudentRecords/wsdl" />
      </operationIdentifier>
    </ACLMessageContentEnvelope>
  </content>
</ACLMessage>
```

❶ This ACL message's fields are used by the agents to perform inter-agent messaging. The *performative* indicates the basic type of the message. The *sender* and *receiver* fields identify the sender and receiver of the message through the use of agent *URIs*. The value of the *conversation-id* uniquely identifies the conversation within the sender agent; it may be used to correlate subsequent failure notification messages from that sender agent or be included in such notification messages sent from the receiver to the sender. The value of the *protocol* field is the identifier for the workflow script being executed by the sender agent. Finally, the *in-reply-to* and *reply-with* fields are absent, because this message is not part of a WSDL request-response operation.

❷ From the point of view of the Instructor Agent, this operation identifier identifies the operation that begins the student record conversation with the Department

Agent. Unfortunately, while the operation is still used in the updated workflow of the Department Agent, it no longer begins the conversation.

Example 4.2. Initial Student Record Request Message

Upon detection of this conversation failure, the Department Agent sends a *not-understood* failure notification message to the Instructor Agent, which can be seen in Example 4.3, “Conversation Failure Signal Message”. A *Conversation Failure Resolution* intention is created to resolve the conversation failure. This intention consults the authority policy to determine the authority agent for the “Student Record Provider” workflow script in cases of “unknown conversation” failure (which, recall, is a subtype of the “unexpected message” failure type). The policy identifies the Department Agent as the authority for all types of failure, and as no updating is therefore necessary, the intention completes without further action.

```
<?xml version="1.0" encoding="utf-8"?>
<ACLMessage conversationID="ProcessSpawningIntention.122"
            performative="not-understood"
            receiverAgentNamespace="http://…/wrabbit/examples/InstructorAgent"
            senderAgentNamespace="http://…/wrabbit/examples/DepartmentAgent">
  <content>
    <ACLFailureEnvelope>
      <failure>
        <unknownConversationFailure originatorNamespace="http://…/wrabbit/examples/DepartmentAgent" />❶
      </failure>
      <failureCause>
        <ACLMessage conversationID="GetMessageContentIntention.120;WorkflowScriptExecutionIntention.121"❷
                    performative="inform"
                    protocol="http://…/wrabbit/examples/ObtainStudentRecords/script"
                    receiverAgentNamespace="http://…/wrabbit/examples/DepartmentAgent"
                    senderAgentNamespace="http://…/wrabbit/examples/InstructorAgent">
          <content>
            <ACLMessageContentEnvelope>
              <messageContent>
                <messageContent>
                  <messageContentEntries>
                    <messageContentEntry>
                      <contentNode>
                        <node>request id 180046</node>
                      </contentNode>
                      <messagePart>
                        <messagePart name="requestID"
                                     namespace="http://…/wrabbit/examples/StudentRecords/wsdl"
                                     type="string" />
                      </messagePart>
                    </messageContentEntry>
                    <messageContentEntry>
                      <contentNode>
                        <node>student record id goes here</node>
                      </contentNode>
                      <messagePart>
                        <messagePart name="studentRecordID"
                                     namespace="http://…/wrabbit/examples/StudentRecords/wsdl"
                                     type="string" />
                      </messagePart>
                    </messageContentEntry>
                    <messageContentEntry>
                      <contentNode>
                        <node>http://…/wrabbit/examples/InstructorAgent</node>
                      </contentNode>
                      <messagePart>
                        <messagePart name="endpointReference"
                                     namespace="http://…/wrabbit/examples/StudentRecords/wsdl"
                                     type="string" />
                      </messagePart>
                    </messageContentEntry>
                  </messageContentEntries>
                </messageContent>
              </messageContent>
              <operationIdentifier>
                <operationIdentifier operationName="requestRecords"
                                     portTypeName="requestAcceptorPT"
```

```
                                    portTypeNamespace="http://…/wrabbit/examples/StudentRecords/wsdl"
                                    wsdlDocumentNamespace="http://…/wrabbit/examples/StudentRecords/wsdl" />
          </operationIdentifier>
        </ACLMessageContentEnvelope>
      </content>
    </ACLMessage>
  </failureCause>
  </ACLFailureEnvelope>
 </content>
</ACLMessage>
```

❶ The *not-understood* message includes a description of the particular failure it is describing. In this case, it is an "unknown conversation" failure.

❷ This message also includes the message that caused the conversation failure to occur. This includes the original message's conversation identifier, which can be used by the notified agent to terminate ongoing script execution.

Example 4.3. Conversation Failure Signal Message

The *not-understood* message, upon its arrival at the Instructor Agent, will be mapped to the *Workflow Script Execution* intention that is executing the "Student Record Obtainer" script. This is done through the use of the value of the *conversation-id* field of the message that caused the failure to occur, included in the *not-understood* message. Since this value identifies the intention (which is waiting for the student records message from the Department Agent), the script execution is terminated. Because the Department Agent is the Instructor Agent's only partner in the "Student Record Obtainer" script, no execution failure notification messages are sent. The *Get Typed Value* intention that created the *Workflow Script Execution* intention will notice its termination, and will create a *Conversation Failure Resolution* intention to resolve the failure. Again, the policy is consulted, and the Department Agent identified as the authoritative source for the "Student Record Obtainer" workflow script. Thus, the *Conversation Failure Resolution* intention sends a series of requests for the documents that define the workflow script to the Department Agent. This exchange itself is done through the use of workflow scripts, and therefore the messages exchanged will not be examined, because such an examination would add little to this discussion. Once all the documents have been obtained, the *Get Typed Value* intention will use the workflow script composition algorithm to obtain an executable composed script, and then create a new *Workflow Script Execution* intention to execute it. As described above, the composed script now features the "Authorization Token Retrieval" script to satisfy the new need for an authorization token.

When the intention begins the execution of the "Student Record Obtainer" script, it encounters the information value need, and pauses the execution. It then determines that the "Authorization Token Retrieval" script is the provider of this value in the composed script, and begins its execution. In pursuing this course of action, the Instructor Agent will begin by sending two messages that constitute the Instructor Agent's request for an authorization token. The Department Agent will then respond with the message that contains the requested authorization token. The receipt of this message will complete the Instructor Agent's execution of the "Authorization Token Retrieval" script.

The first of the two messages the Instructor Agent sends to request an authorization token is listed in Example 4.4, "Authorization Exchange Start Message". The reason for the pair of messages is that the exchange is derived from a WSDL request-response operation, and thus is implemented in ACL messaging using the method summarized in Table 3.1, "Mapping of BPEL activities to ACL message protocols". The ACL mapping also makes use of additional message fields to provide correlation between the messages involved in the operation. As this is the first message in the operation, the value of its *reply-with* field will be used to associate the other messages that make up the operation with the conversation. The value is recorded by the *Workflow Script Execution* intention that the Department Agent creates in response to this message (after having successfully routed it to its *Script Execution Spawning* intention responsible for beginning execution of the "Authorization Token Disbursement" script).

```
<?xml version="1.0" encoding="utf-8"?>
<ACLMessage conversationID="GetMessageContentIntention.120;WorkflowScriptExecutionIntention.145"
            performative="inform"
            protocol="http://…/wrabbit/examples/ObtainAuthorization/script"
            receiverAgentNamespace="http://…/wrabbit/examples/DepartmentAgent"
            replyWith="GetMessageContentIntention.120;WorkflowScriptExecutionIntention.145|1"❶
            senderAgentNamespace="http://…/wrabbit/examples/InstructorAgent">
  <content>
    <ACLMessageContentEnvelope>
      <messageContent>
        <messageContent>
          <messageContentEntries>
            <messageContentEntry>
              <contentNode>
                <node>request id 188261</node>❷
              </contentNode>
              <messagePart>
                <messagePart name="requestID"
                             namespace="http://…/wrabbit/examples/Authorization/wsdl"
                             type="string" />
              </messagePart>
            </messageContentEntry>
          </messageContentEntries>
        </messageContent>
      </messageContent>
      <operationIdentifier>❸
        <operationIdentifier operationName="provideAuthorizationOp"
                             portTypeName="authorizationProviderPT"
                             portTypeNamespace="http://…/wrabbit/examples/Authorization/wsdl"
                             wsdlDocumentNamespace="http://…/wrabbit/examples/Authorization/wsdl" />
      </operationIdentifier>
    </ACLMessageContentEnvelope>
  </content>
</ACLMessage>
```

❶ The value of the *reply-with* field will be matched with that of the subsequent *request* message, allowing it to be associated with the conversation that this message initiates.

❷ This application data would be a good candidate for a correlation token. However, because there is only one WSDL operation used in the authorization workflow, no BPEL message correlation is required.

❸ The invocation of this WSDL request-response operation, which is accomplished through this *inform* message, begins the execution of the "Authorization Token Disbursement" script.

Example 4.4. Authorization Exchange Start Message

The second of the two messages the Instructor Agent sends to request an authorization token is listed in Example 4.5, "Authorization Exchange Request Message". It is a *request* message that shares the value of its *reply-with* field with its predecessor. This value is compared to the previous value by the *Workflow Script Execution* intention that is executing the "Authorization Token Disbursement" script, and since these match, the message is associated with the conversation. Having sent both required messages, the Instructor Agent pauses to wait for the Department Agent to send its response.

```
<?xml version="1.0" encoding="utf-8"?>
<ACLMessage conversationID="GetMessageContentIntention.120;WorkflowScriptExecutionIntention.145"
            performative="request"
            protocol="http://…/wrabbit/examples/ObtainAuthorization/script"
            receiverAgentNamespace="http://…/wrabbit/examples/DepartmentAgent"
            replyWith="GetMessageContentIntention.120;WorkflowScriptExecutionIntention.145|1"❶
            senderAgentNamespace="http://…/wrabbit/examples/InstructorAgent">
  <content>
    <ACLMessageContentEnvelope>
      <messageContent />
      <operationIdentifier>
        <operationIdentifier operationName="provideAuthorizationOp"
                             portTypeName="authorizationProviderPT"
                             portTypeNamespace="http://…/wrabbit/examples/Authorization/wsdl"
                             wsdlDocumentNamespace="http://…/wrabbit/examples/Authorization/wsdl" />
      </operationIdentifier>
    </ACLMessageContentEnvelope>
  </content>
</ACLMessage>
```

❶ Since this is a *request* message, it is obviously part of a WSDL request-response operation, and as such features a *reply-with* field. The value of this field is matched with that of the previous *inform* message, thus associating this message with the conversation begun by that message.

Example 4.5. Authorization Exchange Request Message

The Department Agent responds to the Instructor Agent's request with the message listed in Example 4.6, "Authorization Exchange Response Message". This is the last in the sequence of ACL messages that constitute the WSDL request-response operation.

```
<?xml version="1.0" encoding="utf-8"?>
<ACLMessage conversationID="ProcessSpawningIntention.123;WorkflowScriptExecutionIntention.146"
            inReplyTo="GetMessageContentIntention.120;WorkflowScriptExecutionIntention.145|1"❶
            performative="inform"
            protocol="http://…/wrabbit/examples/ProvideAuthorization/script"
            receiverAgentNamespace="http://…/wrabbit/examples/InstructorAgent"
            senderAgentNamespace="http://…/wrabbit/examples/DepartmentAgent">
  <content>
    <ACLMessageContentEnvelope>
      <messageContent>
        <messageContent>
          <messageContentEntries>
            <messageContentEntry>
              <contentNode>
                <node>request id 188261</node>
              </contentNode>
              <messagePart>
                <messagePart name="requestID"
                             namespace="http://…/wrabbit/examples/Authorization/wsdl"
                             type="string" />
              </messagePart>
            </messageContentEntry>
            <messageContentEntry>
              <contentNode>
                <node>a fake authorization token</node>
              </contentNode>
              <messagePart>
                <messagePart name="authorizationToken"
                             namespace="http://…/wrabbit/examples/Authorization/wsdl"
                             type="string" />
              </messagePart>
            </messageContentEntry>
          </messageContentEntries>
        </messageContent>
      </messageContent>
      <operationIdentifier>❷
        <operationIdentifier operationName="provideAuthorizationOp"
                             portTypeName="authorizationProviderPT"
                             portTypeNamespace="http://…/wrabbit/examples/Authorization/wsdl"
                             wsdlDocumentNamespace="http://…/wrabbit/examples/Authorization/wsdl" />
      </operationIdentifier>
    </ACLMessageContentEnvelope>
  </content>
</ACLMessage>
```

❶ This message is the response portion of a WSDL request-response operation, and thus includes an *in-reply-to* field. The value of this field is taken from the *reply-with* fields of the pair of messages that comprised the request portion of the operation.

❷ The operation identifier included in this message does not identify any WSDL operation provided by the Instructor Agent, a condition unique to response *inform* messages.

Example 4.6. Authorization Exchange Response Message

The authorization token thus obtained, the Instructor Agent returns to the execution of the “Student Record Obtainer” script, providing the required token. This allows the agent to send out the added message, which features the token, to the Department Agent. It can be seen in Example 4.7, “Student Record Authorization Message”. The agent then sends a student record request message similar to the one it had sent earlier, which is listed in Example 4.8, “Student Record Selection Message”. Finally, the Department Agent replies with the student record message, featured in Example 4.9, “Student Record Response Message”.

The first message in this sequence is the *inform* message sent by the Instructor Agent that contains the authorization token. This is the first message exchange in the student record workflow, and thus is responsible for the creation of a *Workflow Script Execution* intention by the *Script Execution Spawning* intention configured to begin executions of the “Student Record Provider” script. The message features a request identifier in its contents, and the application designers have selected this message part as a correlation token. Thus, its value is retained to determine the conversation membership of subsequent messages.

```
<?xml version="1.0" encoding="utf-8"?>
<ACLMessage conversationID="GetMessageContentIntention.120;WorkflowScriptExecutionIntention.145"
            performative="inform"
            protocol="http://…/wrabbit/examples/ObtainStudentRecords/script"
            receiverAgentNamespace="http://…/wrabbit/examples/DepartmentAgent"
            senderAgentNamespace="http://…/wrabbit/examples/InstructorAgent">
  <content>
    <ACLMessageContentEnvelope>
      <messageContent>
        <messageContent>
          <messageContentEntries>
            <messageContentEntry>
              <contentNode>
                <node>request id 189663</node>❶
              </contentNode>
              <messagePart>
                <messagePart name="requestID"
                             namespace="http://…/wrabbit/examples/StudentRecords/wsdl"
                             type="string" />
              </messagePart>
            </messageContentEntry>
            <messageContentEntry>
              <contentNode>
                <node>a fake authorization token</node>
              </contentNode>
              <messagePart>
                <messagePart name="authorizationToken"
                             namespace="http://…/wrabbit/examples/StudentRecords/wsdl"
                             type="string" />
              </messagePart>
            </messageContentEntry>
          </messageContentEntries>
        </messageContent>
      </messageContent>
      <operationIdentifier>
        <operationIdentifier operationName="authorizeRequestForRecords"
                             portTypeName="authorizationAcceptorPT"
                             portTypeNamespace="http://…/wrabbit/examples/StudentRecords/wsdl"
                             wsdlDocumentNamespace="http://…/wrabbit/examples/StudentRecords/wsdl" />
      </operationIdentifier>
    </ACLMessageContentEnvelope>
  </content>
</ACLMessage>
```

❶ The value of this WSDL message part is identified as a correlation token by the WSDL and BPEL specifications that make up the workflow scripts that define the behaviour of both parties. Its value is used to initialize the correlation set, so that subsequent messages can be identified as belonging to the same conversation.

Example 4.7. Student Record Authorization Message

The second message in this sequence is also sent by the Instructor Agent to the Department Agent, and is the actual request for student records that was previously the cause of conversation failure. It also features a request identifier in the application-level message content, identified as a correlation token, the value of which matches that used in the previous message. This allows the Department Agent's intention to successfully identify the message as part of an ongoing conversation that it is managing. The message also includes information identifying to which agent the student records should be sent. This pattern, often seen in web service usage examples, is called a 'callback', and in BPEL is implemented through the inclusion of a WS-Addressing specification in the sent message, which the receiver uses to send its response. In the WRABBIT system, WS-Addressing is not yet used. Instead, the Instructor Agent provides its identifier in the record request message. Once this message is accepted by the Department Agent's intention, its execution proceeds, as it does not need to wait for further messages, while meanwhile the Instructor Agent's intention has paused until further messages arrive.

```
<?xml version="1.0" encoding="utf-8"?>
<ACLMessage conversationID="GetMessageContentIntention.120;WorkflowScriptExecutionIntention.145"
            performative="inform"
            protocol="http://…/wrabbit/examples/ObtainStudentRecords/script"
            receiverAgentNamespace="http://…/wrabbit/examples/DepartmentAgent"
            senderAgentNamespace="http://…/wrabbit/examples/InstructorAgent">
  <content>
    <ACLMessageContentEnvelope>
      <messageContent>
        <messageContent>
          <messageContentEntries>
            <messageContentEntry>
              <contentNode>
                <node>request id 189663</node>❶
              </contentNode>
              <messagePart>
                <messagePart name="requestID"
                             namespace="http://…/wrabbit/examples/StudentRecords/wsdl"
                             type="string" />
              </messagePart>
            </messageContentEntry>
            <messageContentEntry>
              <contentNode>
                <node>student record id goes here</node>
              </contentNode>
              <messagePart>
                <messagePart name="studentRecordID"
                             namespace="http://…/wrabbit/examples/StudentRecords/wsdl"
                             type="string" />
              </messagePart>
            </messageContentEntry>
            <messageContentEntry>
              <contentNode>
                <node>http://…/wrabbit/examples/InstructorAgent</node>❷
              </contentNode>
              <messagePart>
                <messagePart name="endpointReference"
                             namespace="http://…/wrabbit/examples/StudentRecords/wsdl"
                             type="string" />
              </messagePart>
            </messageContentEntry>
          </messageContentEntries>
        </messageContent>
       </messageContent>
       <operationIdentifier>
         <operationIdentifier operationName="requestRecords"
                              portTypeName="requestAcceptorPT"
                              portTypeNamespace="http://…/wrabbit/examples/StudentRecords/wsdl"
                              wsdlDocumentNamespace="http://…/wrabbit/examples/StudentRecords/wsdl" />
       </operationIdentifier>
     </ACLMessageContentEnvelope>
   </content>
 </ACLMessage>
```

❶ The value of this WSDL message part is also identified as a correlation token, and its value matches that used in the previous message. Thus, the Department Agent is able to associate this message with the ongoing conversation.

❷ The agent's identifier is used here to indicate to whom to provide the student record. While in a complete BPEL execution engine, this would be a WS-Addressing specification, this level of detail is unnecessary for this work.

Example 4.8. Student Record Selection Message

The third message in this sequence is the Department Agent's reply to the request sent by the Instructor Agent, and contains the requested student record. This message contains the same request identification token in its application-level message content that the other two messages contained, which allows the Instructor Agent to correlate this message with its ongoing conversation. Once this message is sent, the Department Agent's script is complete, and once the message is received, the Instructor Agent's script is also complete. Thus, the conversation is successful.

```
<?xml version="1.0" encoding="utf-8"?>
<ACLMessage conversationID="ProcessSpawningIntention.122;WorkflowScriptExecutionIntention.147"
            performative="inform"
            protocol="http://…/wrabbit/examples/ProvideStudentRecords/script"
            receiverAgentNamespace="http://…/wrabbit/examples/InstructorAgent"
            senderAgentNamespace="http://…/wrabbit/examples/DepartmentAgent">
  <content>
    <ACLMessageContentEnvelope>
      <messageContent>
        <messageContent>
          <messageContentEntries>
            <messageContentEntry>
              <contentNode>
                <node>request id 189663</node>❶
              </contentNode>
              <messagePart>
                <messagePart name="requestID"
                             namespace="http://…/wrabbit/examples/StudentRecords/wsdl"
                             type="string" />
              </messagePart>
            </messageContentEntry>
            <messageContentEntry>
              <contentNode>
                <node>this is a student record</node>
              </contentNode>
              <messagePart>
                <messagePart name="studentRecords"
                             namespace="http://…/wrabbit/examples/StudentRecords/wsdl"
                             type="string" />
              </messagePart>
            </messageContentEntry>
          </messageContentEntries>
        </messageContent>
      </messageContent>
      <operationIdentifier>
        <operationIdentifier operationName="receiveRequestedRecords"
                             portTypeName="recordAcceptorPT"
                             portTypeNamespace="http://…/wrabbit/examples/StudentRecords/wsdl"
```

```
                              wsdlDocumentNamespace="http://…/wrabbit/examples/StudentRecords/wsdl" />
      </operationIdentifier>
    </ACLMessageContentEnvelope>
  </content>
</ACLMessage>
```

❶ Once again, this application-level token is used to associate this message with the ongoing conversation, this time by the Instructor Agent.

Example 4.9. Student Record Response Message

4.3. Changes to Partner Restrictions Case Study

This case study involves a workflow featuring the Department Agent, the Teaching Assistant Agent (or TA Agent for short) and the Instructor Agent. The conversation failure occurs between the Department and TA Agents, and is due to a partner restriction loosening performed at the Instructor Agent and distributed to the TA Agent.

4.3.1. Case Study Details

This study uses the "Student Record" workflow featured in Section 4.2, "Missing Precondition for Introduced Exchange Case Study", and diagrammed in Figure 4.1, "Workflow Execution before Added Exchange". Unlike in the previous case study, however, authorization tokens are not required. Rather, in the initial case, the Department Agent is only allowed to send student records to the Instructor Agent. This is enforced through the use of a restriction on the agents who may become the "student record destination" partner in the Department Agent's "Student Record Provider" script. However, the Instructor Agent loosens this restriction to also allow the TA Agent to perform this function, allowing it to delegate some work to that agent. The modified files that make up this new script are made available to the TA Agent, making it aware that it is now a legitimate partner in the student record workflow.

4.3.2. Case Study Exchange

To obtain a student record, the TA Agent selects the "Student Record Obtainer" workflow script, which is non-abstract and produces a student record message, and begins this script's execution. The script features a conversation with the Department Agent, and so the TA Agent sends a request to obtain a student record to the Department

Agent. However, only the Instructor Agent and the TA Agent are aware of the workflow script modification, and thus the Department Agent expects the requester of transcripts to be the Instructor Agent. This constitutes a conversation failure of the type "illegal partner", and therefore the Department Agent sends a *not-understood* failure notification message to the TA Agent.

The resolution of this mismatch requires all three agents, even though the Instructor Agent was not part of the original conversation. The authority policy specifies that by default the Department Agent is the authority on the "Student Record Provider" and "Student Record Obtainer" workflow scripts. However, a failure-specific policy exists for the "illegal partner" failure type, and indicates that the Instructor Agent is the authority for these scripts. Therefore, both the Department Agent and the TA Agent determine that the Instructor Agent is the authoritative source for their respective scripts, and both obtain the documents from which their scripts are derived from the Instructor Agent. These include the updated partner restriction files that incorporate the modification. Once the new script is retrieved, the TA Agent reinitiates the request for student records (in the event that the Department Agent has not completed its update, it queues the request message) and the conversation completes successfully.

4.4. Reordered Message Exchanges Case Study

This case study involves a workflow featuring the Department Agent and the Payroll Agent. The conversation failure that occurs between these agents is due to a reordering of the messages exchanged that was done by the developer of the Payroll Agent.

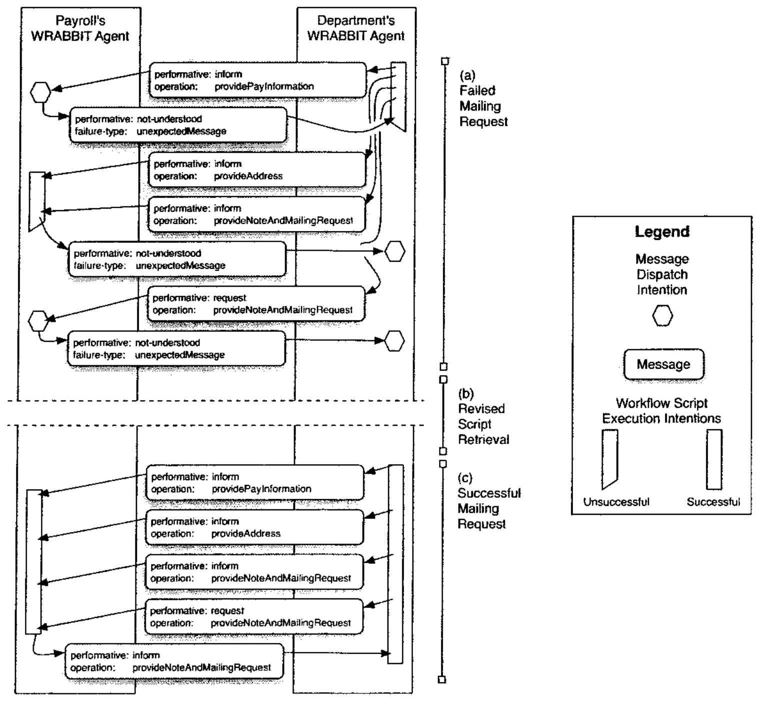
4.4.1. Case Study Details

The Payroll Agent provides a service to mail paycheques to employees. This service is provided using the "Employee Paycheque Mailing" workflow script, which requires messages containing the employee's salary information, the address of the employee, and a note to print on the paycheque (for example, a seasonal greeting, or a reminder). Once each of these messages have been received in that order, the agent then returns a message containing the estimated time of arrival for the paycheque. The note message and the arrival estimate message are implemented using a WSDL request-response operation, and are thus implemented with three messages as prescribed by Table 3.1, "Mapping of BPEL activities to ACL message protocols". The actual work of printing and mailing the cheque is performed by the script through the invocation of an existing web service. The script is designed to work with a single partner, whose complimentary behaviour is modelled in the "Mail Employee Paycheque" script.

Initially, the Payroll Agent required the contents of the three messages solely to pass them on to the private paycheque printing service. However, to encourage the cheaper method of directly depositing wages, the University decided to charge those who chose to continue using the paper-based method to pay their own postage. Postage varies depending on where the employee is located, so the address is required prior to the deduction to determine the proper amount. Thus, the address message is now required first, as it is used to calculate the cost to be deducted, followed by the pay information message that is used to deduct the appropriate amount. The paycheque note remains the last message received by the Payroll Agent. The three messages are then used to print and mail the paycheque after the deduction, as before.

4.4.2. Case Study Exchange

In this case study, the Department Agent is configured with the earlier version of the "Mail Employee Paycheque" script, and is thus unaware of this service modification. It executes the script in order to send a paycheque to an employee, and in accordance with its current interaction model with the Payroll Agent, sends the pay information first. The exchange is diagrammed in Figure 4.3, "Workflow Execution with Reordered Message Exchange". The Department Agent's initial pay information message is expected by the Payroll Agent as the second message, and so the agent's response is a *not-understood* message (Figure 4.3(a)). The failure type here is "unexpected message", because the message was not expected at the time it was received. However, before the Department Agent receives and processes this *not-understood* message, it continues to send out messages according to its (out of sync) conversation script. Both the address information and note message exchanges are processed by the Department Agent, and since the note message exchange is part of a WSDL request-response operation, three additional messages are sent to the Payroll Agent. Upon receipt of the *not-understood* message from the Payroll Agent, the Department Agent routes it to the ongoing conversation, and halts the execution of the script with which it was associated. Meanwhile, the Payroll Agent recognizes the Department Agent's second message, which contains the address information, as the message that starts the execution of the updated "Employee Paycheque Mailing" script, which it is configured to execute on demand. Of course, the Department Agent's third message, containing the note to be printed on the paycheque, is not anticipated by the Payroll Agent, which expected the second message to contain address information. Thus, the third and fourth messages also receive replies of not-understood (Figure 4.3(a)).



Payroll's WRABBIT Agent
Department's WRABBIT Agent
performative: inform
operation: providePayInformation
performative: not-understood
failure-type: unexpectedMessage
performative: inform
operation: provideAddress
performative: inform
operation: provideNoteAndMailingRequest
performative: not-understood
failure-type: unexpectedMessage
performative: request
operation: provideNoteAndMailingRequest
performative: not-understood
failure-type: unexpectedMessage
performative: inform
operation: providePayInformation
performative: inform
operation: provideAddress
performative: inform
operation: provideNoteAndMailingRequest
performative: request
operation: provideNoteAndMailingRequest
performative: inform
operation: provideNoteAndMailingRequest
(a) Failed Mailing Request
(b) Revised Script Retrieval
(c) Successful Mailing Request
Legend
Message Dispatch Intention
Message
Workflow Script Execution Intentions
Unsuccessful
Successful

Figure 4.3. Workflow Execution with Reordered Message Exchange

The modification of the workflow therefore caused three separate conversation failures to occur over the course of this interaction. Each of these failures will cause the conversation error resolution process to begin. When the agents consult the shared authority policy, they discover that for this type of error ("unexpected message") and the concerned workflow scripts ("Mail Employee Paycheque" and "Employee Paycheque Mailing"), the Payroll Agent is the authority. Thus, the Payroll Agent does not need to take any further action. The Department Agent, on the other hand, will need to obtain the component documents that make up the "Mail Employee Paycheque" workflow script (Figure 4.3(b)). Because of an optimization in the WRABBIT system, only one of the three recovery processes will actually fetch the updated documents. Once the Department Agent has the up-to-date script, it will again try to satisfy its need for the Payroll Agent's return message. The "Mail Employee Paycheque" script will once again be selected and executed, and in this instance will succeed, because the workflow scripts are compatible (Figure 4.3(c)).

4.5. Deleted Message Exchange Case Study

Both the Instructor Agent and the Department Agent are featured in this case study. The conversation between these agents fails because one of the message exchanges between them is deleted in the Department Agent's script.

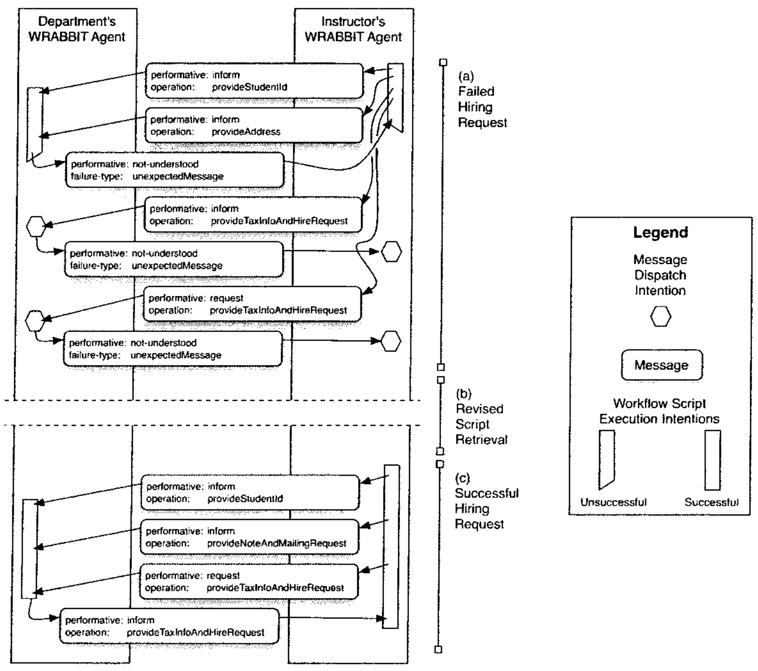
4.5.1. Case Study Details

In addition to the student record workflow mentioned above, the Department Agent also provides a service to employ students (for example, as summer research assistants). Prior to the modification, the "Employ Student" script used by the Instructor Agent to employ a student specified that messages containing the student's identification number, the student's address, and the student's tax information, would have to be sent to the Department Agent. In turn, its script, "Make Student Employee", would then use a web service to create an employee record in the employee database, and would return the employee identification number. The tax information and employee identifier messages are implemented using a WSDL request-response operation, and are thus implemented with three ACL messages as prescribed by Table 3.1, "Mapping of BPEL activities to ACL message protocols", while the others are one-way message exchanges.

In an effort to reduce duplication and out-of-date information, the University has introduced a consolidated database for a person's contact information, and has modified existing databases to refer to this centralized data store. Because a student will already have an address record in this new database, the address is no longer required when creating the student's employee database record, as it will simply include a reference to this information taken from the student's record. Thus, the message containing the address is removed from the workflow.

4.5.2. Case Study Exchange

Once again, the Instructor Agent is not notified of the modification, and thus when attempting to employ a student, will include the student's identifier, address, and tax information in the four messages it sends to the Department Agent, as shown in Figure 4.4, "Workflow Execution with Deleted Message Exchange". The first message in the conversation is expected by the Department Agent, and so it begins the execution of the "Make Student Employee" script. However, the arrival of the second message is a conversation failure of "unexpected message" type, and as such, the Department Agent terminates the ongoing execution, and issues a *not-understood* failure notification message. As a result, the remaining two messages will not belong to any ongoing conversation, and will thus also cause "unexpected message" failures. When the Instructor Agent receives the first *not-understood* message, it will abandon its execution of the "Employ Student" script, which was paused while waiting for the Department Agent's reply (Figure 4.4(a)). After the reception of the other two *not-understood* messages, the Instructor Agent will be notified of three conversation failures that require recovery.



Department's WRABBIT Agent
Instructor's WRABBIT Agent
performative: inform
operation: provideStudentId
performative: inform
operation: provideAddress
performative: not-understood
failure-type: unexpectedMessage
performative: inform
operation: provideTaxInfoAndHireRequest
performative: not-understood
failure-type: unexpectedMessage
performative: request
operation: provideTaxInfoAndHireRequest
performative: not-understood
failure-type: unexpectedMessage
performative: inform
operation: provideStudentId
performative: inform
operation: provideNoteAndMailingRequest
performative: request
operation: provideTaxInfoAndHireRequest
performative: inform
operation: provideTaxInfoAndHireRequest
(a) Failed Hiring Request
(b) Revised Script Retrieval
(c) Successful Hiring Request
Legend
Message Dispatch Intention
Message
Workflow Script Execution Intentions
Unsuccessful
Successful

Figure 4.4. Workflow Execution with Deleted Message Exchange

To resolve these three errors, the agents consult the shared authority policy. Here, the Department Agent is identified as the authoritative source for both scripts. The Department Agent therefore resolves the three failures without any action. The Instructor Agent, however, must request the documents that specify the "Employ Student" script from its partner (Figure 4.4(b)). As previously, because of an optimization in the WRABBIT system, only one of the three failure recovery processes will actually fetch the updated documents. Upon successful completion of the recovery processes, the Instructor Agent will once again try to satisfy its need for an employee identification number, and select the "Employ Student" script to do so. In this instance, its execution will succeed, because the workflow scripts are compatible (Figure 4.4(c)).

4.6. Modified Message Exchange Case Study

The two participants in this study are the Department Agent and the Communication Agent (that offers services relating to the University's external communications division). Their conversation fails because of a modified message exchange introduced at the Communication Agent.

4.6.1. Case Study Details

The University's external communication division is responsible for composing letters notifying students of any scholarships that have been awarded to them. Thus, the Communication Agent provides a service for notifying students of scholarships that are awarded by the different levels of organization at the University. This service initially required the scholarship information (such as the name and amount) and also the student's address. The Communication Agent would then return a message containing an estimate of when the letter would be sent. As in previous studies, the last two messages make up a WSDL request-response operation, and are thus implemented with three ACL messages as prescribed by Table 3.1, "Mapping of BPEL activities to ACL message protocols", while the others are one-way message exchanges.

However, in an effort to save money and decrease environmental damage, the University allows individuals to specify their preferred communication medium: electronic or paper mail. This preference is stored in the centralized contact information database mentioned in previous studies. To take advantage of this new ability, the

"Student Scholarship Notification" script that defines the Communication Agent's behaviour was changed such that a message containing the scholarship recipient's identification number is required instead of the message containing the address. The complimentary "Notify Student of Scholarship" script is also changed, but is not distributed to the Department Agent, who uses it whenever the department awards its scholarships.

4.6.2. Case Study Exchange

Since the Department Agent is not aware of the modification, it will execute the previous version of the "Notify Student of Scholarship" script. Therefore, the message exchange between the two agents diagrammed in Figure 4.5, "Workflow Execution with Modified Message Exchange", will include the message with scholarship information, the message containing the address of the scholar, and the request for a reply. The Communication Agent will begin the execution of its "Student Scholarship Notification" script in response to the message containing scholarship information, as it considers this to be the conversation-starting message. However, the second message sent is not that expected by the agent (as it has been replaced in the new version), and so the script execution is terminated and a *not-understood* failure notification message is sent indicating an "unexpected message" failure. As a consequence, the third message has no conversation with which to be associated, and thus also causes a conversation failure, signalled by a *not-understood* message (Figure 4.5(a)). When the first *not-understood* message is received by the Department Agent, its executing script will also be terminated. Both agents now have two conversation failures from which to recover.

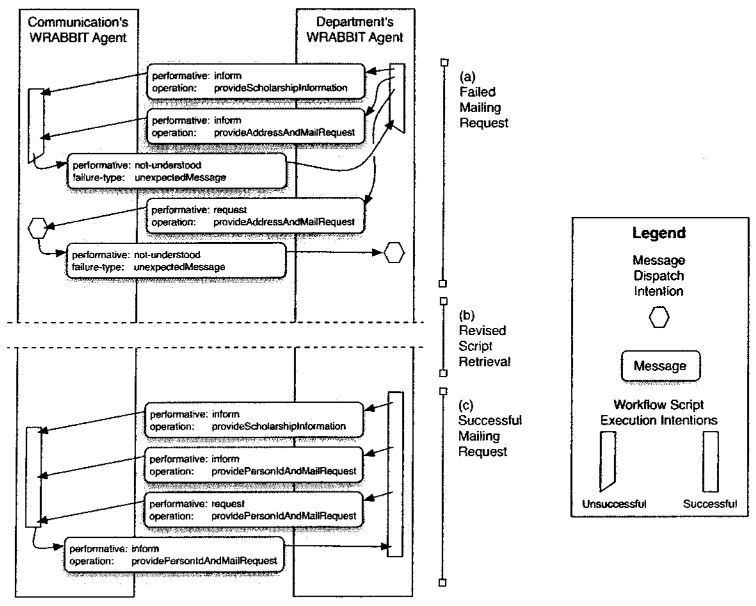

Communication's WRABBIT Agent
Department's WRABBIT Agent
performative: inform
operation: provideScholarshipInformation
performative: inform
operation: provideAddressAndMailRequest
performative: not-understood
failure-type: unexpectedMessage
performative: request
operation: provideAddressAndMailRequest
performative: not-understood
failure-type: unexpectedMessage
performative: inform
operation: provideScholarshipInformation
performative: inform
operation: providePersonIdAndMailRequest
performative: request
operation: providePersonIdAndMailRequest
performative: inform
operation: providePersonIdAndMailRequest
(a) Failed Mailing Request
(b) Revised Script Retrieval
(c) Successful Mailing Request
Legend
Message Dispatch Intention
Message
Workflow Script Execution Intentions
Unsuccessful
Successful

Figure 4.5. Workflow Execution with Modified Message Exchange

The Communication Agent is identified as the authoritative source for both the "Student Scholarship Notification" and "Notify Student of Scholarship" workflow scripts in the declarative policy. Thus, the Communication Agent performs no further recovery actions to resolve the conversation failures. The Department Agent, however, sends requests to the Communication Agent for the documents from which its script is derived (Figure 4.4(b)). As detailed earlier, an optimization in the WRABBIT system ensures that this process is not repeated in response to both failures. With a compatible script now in hand, the Department Agent will resume its effort to notify a student of a scholarship, and will complete the necessary conversation without failures (Figure 4.4(c)).

4.7. Capability and Execution Failures Case Study

This case study includes the Instructor Agent, the Department Agent, and the Teaching Assistant Agent (or TA Agent for short). The three agents initially share a workflow, until the TA Agent is excluded by a partner responsibility reorganization performed at the Department Agent, causing a conversation failure to occur. The conversation failure leads to execution failure notifications, as well as a capability failure during the subsequent execution. The case also includes significant use of the workflow script composition algorithm.

4.7.1. Case Study Details

The Department Agent is capable of modifying a student's grades through its "Student Grades Modification" workflow script. This script has two partners: the first sends a message to the Department Agent containing an authorization token (these tokens are described in Section 4.2). Only the Instructor Agent is allowed to fill this partner's role, as defined by the "Modify Student Grade — Instructor Portion" script. Because it sends an authorization token, but does not produce it, this script is abstract. The second partner sends a message containing the student grade change, and requests that the agent return a message containing a grade change receipt, which it does. This message exchange is captured as a WSDL request-response operation in the "Modify Student Grade — Other Portion" script, which can be executed by any agent. This script is also abstract, requiring the grade change information and the request identifier used to associate this exchange with the corresponding exchange in the "Modify Student Grade — Instructor Portion" script.

Both the Instructor and TA Agents have a "Grade Assignment" workflow script that given an assignment can produce grade change information. This script does not involve any inter-agent communication, only internal web service invocations. Additionally, the Instructor Agent has a "Fetch Assignment" workflow script, that when given an assignment identifier, can produce an assignment. Again, this script does not involve the exchange of any inter-agent messages. Thus, to facilitate the delegation of grading tasks from the Instructor Agent to the TA Agent, the "Student Evaluation" script was created. In response to a message containing the assignment and the request identifier, this script will grade the assignment and submit the grade, returning the grade change receipt in the response message in the WSDL request-response operation. This script is executed by the TA Agent, while its complement, the "Evaluate Student" script, is executed by the Instructor Agent. Both of these scripts are abstract, depending on the workflow script composition algorithm to make them executable.

In the case of the TA Agent's "Student Evaluation" script, the composition algorithm is invoked in response to the arrival of the message containing the request identifier and the assignment. The algorithm constructs the composed script depicted in Figure 4.6, "TA Agent's Composed Script". The "Student Evaluation" script's need for a grade change receipt to return in a message to the Instructor Agent is satisfied by the "Modify Student Grade — Other Portion" script. That script's need for a request identifier can be satisfied by the "Student Evaluation" script at the time its need requires satisfaction, as this script will already have received a message containing the request identifier. Similarly, this script can also provide the assignment to the "Grade Assignment" script so that its execution may satisfy the "Modify Student Grade — Other Portion" script's need for grade change information. Once the agent has constructed such a composite script in response to the initial exchange, it is executed.

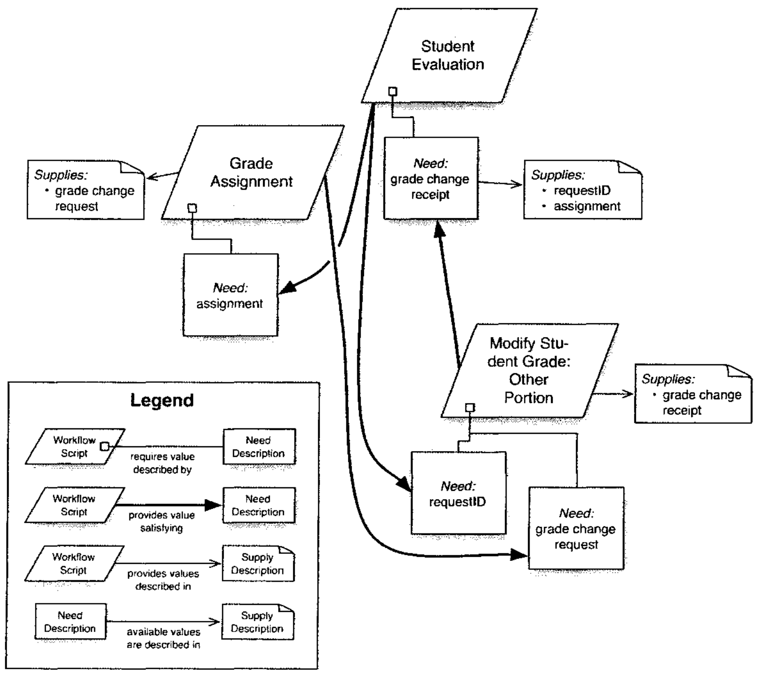

Student Evaluation
Grade Assignment
Supplies: • grade change request
Need: grade change receipt
Supplies: • requestID • assignment
Need: assignment
Modify Student Grade: Other Portion
Supplies: • grade change receipt
Need: requestID
Need: grade change request
Legend
Workflow Script
requires value described by
Need Description
provides value satisfying
provides values described in
Supply Description
available values are described in

Figure 4.6. TA Agent's Composed Script

In this scenario, the Instructor Agent wishes to grade an assignment, and have the resulting grade be input into the department records. It would also prefer to have the TA Agent perform as much of the work as possible. Therefore, when the agent invokes the workflow script composition algorithm with its need for a grade change receipt, it prefers the “Evaluate Student” script to the use of the “Modify Student Grade — Other Portion” and “Grade Assignment” scripts, which it is also capable of executing. The composition is depicted in Figure 4.7, “Instructor Agent’s Composed Script”.

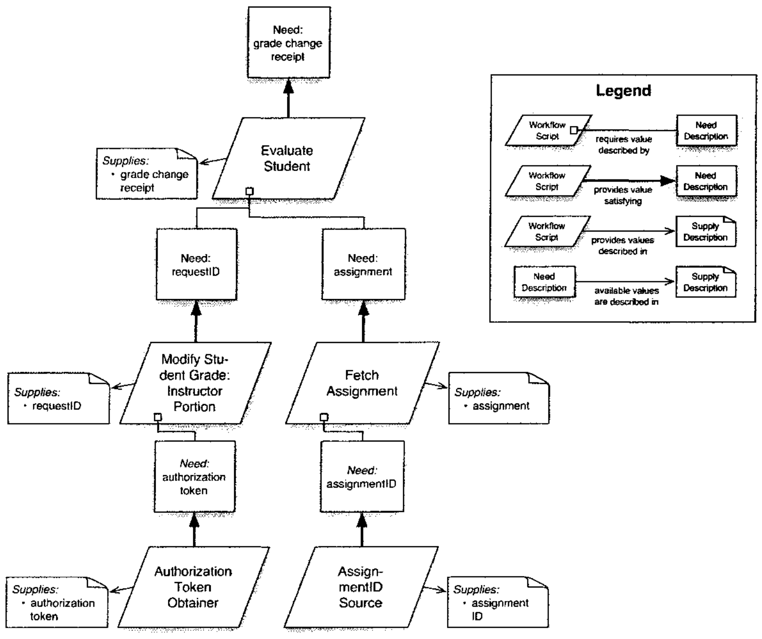

Need: grade change receipt
Supplies: • grade change receipt
Evaluate Student
Need: requestID
Need: assignment
Supplies: • requestID
Modify Student Grade: Instructor Portion
Fetch Assignment
Supplies: • assignment
Need: authorization token
Need: assignmentID
Supplies: • authorization token
Authorization Token Obtainer
Assign-mentID Source
Supplies: • assignment ID
Legend
Workflow Script
requires value described by
Need Description
Workflow Script
provides value satisfying
Need Description
Workflow Script
provides values described in
Supply Description
Need Description
available values are described in
Supply Description

Figure 4.7. Instructor Agent’s Composed Script

When the Instructor Agent executes its composed script, it converses with the other agents as depicted in Figure 4.8, “Workflow Execution before Modification (Abridged)”. (Note that for brevity, the authorization token exchange is not included in the diagram.) As part of these conversations, the TA Agent will use the composed script built around its “Student Evaluation” script to fulfill its obligations. However, this series of successful conversations was made impossible due to a change applied to the student grade modification script.

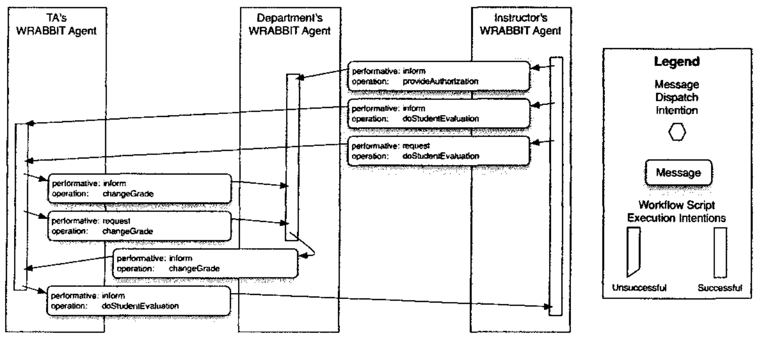


Figure 4.8. Workflow Execution before Modification (Abridged)

In response to a process audit, the department has changed its policy on what agents are allowed to modify student grades, and the TA Agent is no longer included. This is accomplished by modifying the "Student Grades Modification" script such that all of the messages exchanged with the partner whose behaviour was defined by the "Modify Student Grade — Other Portion" script are now exchanged with the partner defined by the "Modify Student Grade — Instructor Portion" script. These scripts are also changed: all of the activities that were previously in the "Modify Student Grade — Other Portion" script are moved into the other script. This reduces the "Modify Student Grade — Other Portion" script to a no-op, while restricting the behaviour it used to define to the Instructor Agent. Neither the TA Agent nor the Instructor Agent is notified of the change.

4.7.2. Case Study Exchange

To begin the study, the Instructor Agent will compose the workflow script found in Figure 4.7, and begin its execution. After having successfully obtained an authorization token through communication with the Department Agent, the agent will execute its unmodified "Modify Student Grade — Instructor Portion" script, and will send a message to the Department Agent. As a consequence, that agent will begin the execution of the revised "Student Grades Modification" script. This sequence is shown in Figure 4.9, "Workflow Execution after Modification (Abridged)", with the exception of the authorization token exchange, omitted for brevity. The Instructor Agent will retrieve an assignment and begin the execution of the "Evaluate Student" script, and as a consequence will send messages to the TA Agent. This will initiate the recipient's construction of a composed workflow script based on its "Student Evaluation" script and diagrammed in Figure 4.6. The execution of this composed script begins with the grading of the assignment, followed by the execution of the unmodified "Modify Student Grade — Other Portion" script. Since it involves a message exchange with the Department Agent, two messages are sent as part of this execution. However, the corresponding script has been modified at the Department Agent, which realizes that the messages it has received are not from the same agent as that that sent the initial message, though from its viewpoint, the same partner is to send both messages. This is a conversation failure of type "illegal partner", which terminates the ongoing execution and causes a *not-understood* failure notification message to be returned to the TA Agent. This termination is also signalled to the Instructor Agent through the use of a *sorry* failure notification message, with execution failure as the particular subtype. Likewise, the TA Agent and the Instructor Agent send out such messages as their script executions are terminated in response to arriving failure notification messages. In Figure 4.9(a), the TA Agent and the Instructor Agent exchange such messages, though the actual ordering of the exchanged *sorry* messages can vary. The second message initially sent by the TA Agent as part of its execution of the outdated "Modify Student Grade — Other Portion" script no longer belongs to an ongoing conversation, and so the Department Agent responds with a *not-understood* message signalling an "unexpected message" failure.

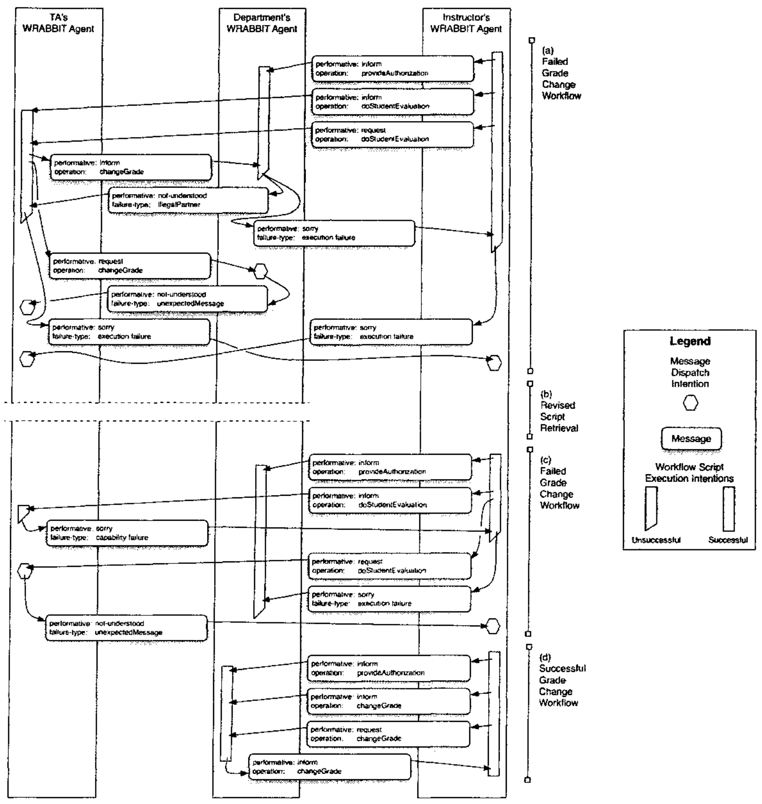

TA's WRABBIT Agent
Department's WRABBIT Agent
Instructor's WRABBIT Agent
(a) Failed Grade Change Workflow
(b) Revised Script Retrieval
(c) Failed Grade Change Workflow
(d) Successful Grade Change Workflow
Legend
Message Dispatch Intention
Message
Workflow Script Execution Intentions
Unsuccessful
Successful

Figure 4.9. Workflow Execution after Modification (Abridged)

The Department Agent is the authority for all types of failure involving the scripts in the student grade modification workflow. Therefore, the TA Agent will obtain the updated version of the files for its "Modify Student Grade — Other Portion" script from the Department Agent (Figure 4.9(b)) as part of the failure resolution process (although, as an optimization, this happens only once between the agent's two failure resolution processes). Meanwhile, the Instructor Agent will restart the execution of its composed script, sending messages once more to the Department and TA Agents (Figure 4.9(c)). Once the TA Agent's recovery process is complete, the first of these messages from the Instructor Agent will once again trigger the workflow script composition algorithm. However, the algorithm will be unable to construct a composed script, as the updated "Modify Student Grade — Other Portion" script does not provide any information values. This failure will be signalled once again with a *sorry* message, with capability failure as the particular failure type. Once received by the Instructor Agent, its script execution is abandoned, triggering an additional *sorry* message notifying the Department Agent of the execution failure. Additionally, the Instructor Agent's "Evaluate Student" script is removed from the pool of scripts available to the composition algorithm, as it is no longer useful. The second message sent by the Instructor Agent to the TA Agent will produce a *not-understood* message, which will be ignored by the Instructor Agent as it refers to a script that it no longer uses. The Instructor Agent then constructs a new composed workflow script using the algorithm. The revised script is pictured in Figure 4.10, "Instructor Agent's Revised Composed Script".

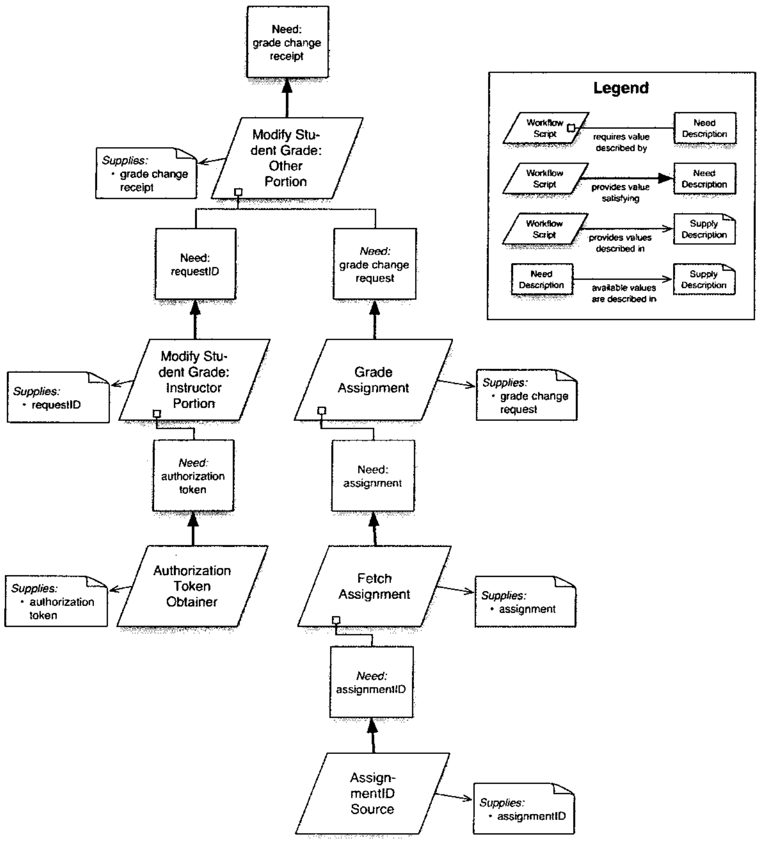

Need: grade change receipt
Modify Student Grade: Other Portion
Supplies: • grade change receipt
Need: requestID
Need: grade change request
Modify Student Grade: Instructor Portion
Supplies: • requestID
Grade Assignment
Supplies: • grade change request
Need: authorization token
Need: assignment
Authorization Token Obtainer
Supplies: • authorization token
Fetch Assignment
Supplies: • assignment
Need: assignmentID
AssignmentID Source
Supplies: • assignmentID
Legend
Workflow Script
requires value described by
Need Description
Workflow Script
provides value satisfying
Need Description
Workflow Script
provides values described in
Supply Description
Need Description
available values are described in
Supply Description

Figure 4.10. Instructor Agent's Revised Composed Script

This new composed workflow script does not involve communication with the TA Agent; the Instructor Agent uses the "Grade Assignment" and "Modify Student Grade — Other Portion" scripts itself. The Instructor's versions of the student grade modification scripts are those missing the modifications. However, when both executed by the

Instructor Agent, the behaviour they prescribe is indistinguishable from that prescribed by the modified scripts, and as such, the Instructor Agent is able to complete its conversation with the Department Agent successfully (Figure 4.9(d)).

4.8. Implementation Capacity Case Study

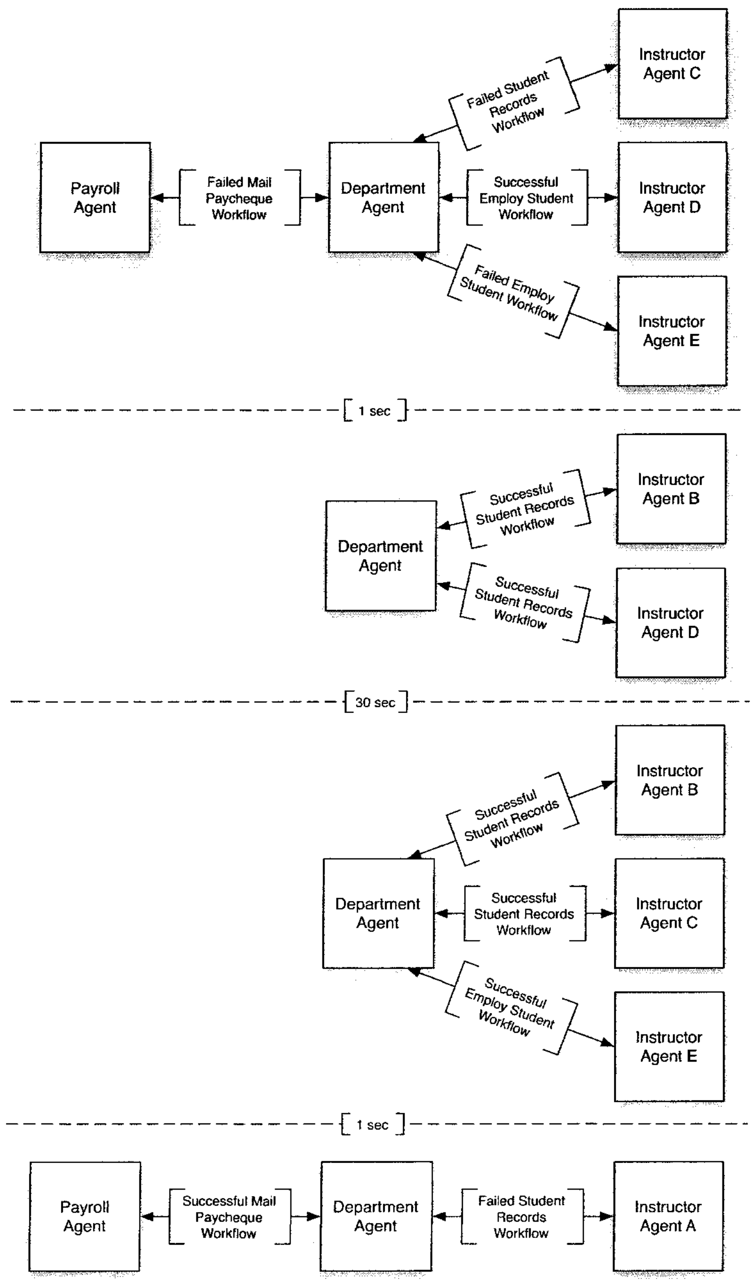
The last case study serves mainly as a demonstration of the state of the implementation of the WRABBIT system. Specifically, it demonstrates that the implementation is capable of handling multiple agents, each executing simultaneously.

4.8.1. Case Study Details

This study is an amalgamation of previous case studies. Specifically, the student record workflow and its modification as presented in Section 4.2, the paycheque mailing workflow and modification of Section 4.4, and the student employment workflow and modification of Section 4.5 are used. In this study, a single Department Agent interacts with a Payroll Agent and five Instructor Agents. Each agent in the study is configured to execute one or more of the workflows, at a particular time and with or without the modification that leads to conversation failures.

4.8.2. Case Study Exchange

The execution of the case study is shown in Figure 4.11, "Workflow Execution Order". As is evident from the diagram, there are four bursts of activity, of which there are two pairs. Each pair executes with a small delay between its component bursts, so that their execution overlaps, while a larger delay exists between the two pairs to ensure the execution of the workflows has completed. This makes the case study a useful test to ensure that the implementation does not rely upon a freshly initialized state to function properly.



Instructor Agent C
Failed Student Records Workflow
Payroll Agent
Failed Mail Paycheque Workflow
Department Agent
Successful Employ Student Workflow
Instructor Agent D
Failed Employ Student Workflow
Instructor Agent E
1 sec
Instructor Agent B
Successful Student Records Workflow
Department Agent
Successful Student Records Workflow
Instructor Agent D
30 sec
Instructor Agent B
Successful Student Records Workflow
Department Agent
Successful Student Records Workflow
Instructor Agent C
Successful Employ Student Workflow
Instructor Agent E
1 sec
Payroll Agent
Successful Mail Paycheque Workflow
Department Agent
Failed Student Records Workflow
Instructor Agent A

Figure 4.11. Workflow Execution Order

Chapter 5. Conclusion

This work has presented the WRABBIT system, an implementation of a plausible solution to the challenge of maintaining a consistent distributed workflow when the workflow scripts that define the operation of each of the constituent endpoints are modified independently. The contributions of this work are presented in Section 5.1, “Contributions”, while the limitations of the proposed solution’s coverage of this problem are examined in Section 5.2, “Problem Coverage”.

5.1. Contributions

The emergence of web services has renewed interest in the automation of inter-organizational workflows, through the use of standards such as BPEL. Modification of inter-organizational workflows requires that the external behaviour of all participants is compatible; in the case of web services, this external behaviour is the exchange of messages. The distributed nature of the workflow’s execution all but requires a corresponding distribution of the specification of all of its behaviour: to do otherwise risks coupling the internal behaviours of the participating organizations. This restriction does not preclude global modelling of external behaviour, as advocated by the incomplete WS-CDL, however, each web service endpoint will require a complete model of its behaviour to guide its execution. This work proposes that this model consist of a set of BPEL specifications, annotated and combined using methods inspired by semantic web service research. This approach allows independent workflow modifications to be applied to the set of workflow scripts that define the behaviour of each participant, which can then be distributed to the respective participant’s endpoint and integrated into the combined script.

The principal contribution of this work is the method by which these workflow modifications are distributed. When the behaviour of one participant in the workflow changes unilaterally, the mismatching models will be identifiable at run-time by messages unexpected by the receiving participant. This work uses agents as BPEL execution engines, which allows these conversation failures to be identified and signalled. Once the agents are made aware of the failures, either through detection or signalling, they can take steps to resolve the underlying cause: mismatched specifications. This error recovery process is defined by a shared policy, which includes a declarative specification relating the failed conversation and the failure type to a source of authoritative behaviour. The agents obtain the specifications that define their respective behaviour from the authoritative source, resolving their conversation failure. One advantage of this approach is isolation of the recovery effort: the error is detected and resolved by the two partners between whom the failed conversation occurred. By distributing updated workflow scripts in this way, the agents also avoid unnecessary distribution that might occur if the agents converse less often than the workflow script is updated, or if only a rarely used section of the conversation between the agents was modified. This method also allows workflow developers to focus their attention on application-level challenges, leaving distribution to automation.

Having proposed this method, this work described an implementation that was constructed to exercise and evaluate the idea. The implemented method was evaluated with case studies of varying complexity. In each of the case studies, a conversation failure affects the agents' interaction, and the proposed method is able to resolve the failure. The agents are thus able to re-execute the workflow without re-encountering the conversation failure. While the set of case studies is limited, they are sufficient to demonstrate that the proposed method works, if only for those situations similar to these cases. Since the demonstration was successful, this work argues that the proposed method is deserving of further study.

5.2. Problem Coverage

The challenge of facilitating independent endpoint behaviour modification within distributed workflows is difficult to address. While this work proposes a method that addresses this challenge, in its current form, it proposes challenges of its own. These challenges are listed in Section 5.2.1, "Limitations of Proposed Solution". Similarly, the implementation of the proposed method also suffers from limitations that prevent a more thorough evaluation of the method to occur. These deficiencies are identified in Section 5.2.2, "Implementation Limitations".

5.2.1. Limitations of Proposed Solution

The proposed method for detecting and resolving conversation errors relies upon the homogeneity of the agents involved in the distributed workflow. While the proposed method is independent of the implementation language used, it requires that the agents act as peers, responding to conversation failures using a shared failure-resolution policy. As such, though the agents interoperate with traditional web services, since the method is applied only in inter-agent conversations, it is only beneficial when widely adopted. This limitation seems unavoidable, and its drawbacks are certainly outweighed by the benefits of the application of this method. However, it is possible that a comprehensive effort to create a complete and unambiguous standard describing the method, rather than its presentation here as part of this work, might increase the likelihood of its adoption.

The proposed method also fails to detect all possible workflow mismatches as conversation failures. As an example, consider a conversation where the last message exchanged between the two agents is removed from the script of the message's sender. In this environment, when the sender agent arrives at that point in its script, it will no longer send the message, as it the exchange is no longer present in the script. Instead, it will continue execution of the script, possibly completing it, depending on the state of its conversations with other partners. The receiver of the message, whose script execution will be paused, will be waiting for the arrival of the message from its partner, and since the partner will not send any further messages in this conversation, the script's execution will not advance. The conversation failure will thus remain undiscovered. This same situation can also occur when an exchange is added in the receiver's model. A different example situation is found in a callback pattern when a message exchange to be sent by the target of a callback exchange just prior to the callback exchange itself is deleted in the target's script. Here, the callback's target will be waiting for the callback message to arrive, while its partner is waiting for the deleted message before proceeding. This situation could also occur through the addition of a message receiving activity prior to the callback message sending activity in the agent's script. While the script developer could include a quality of service constraint to ensure a timely response in these cases, this would be implemented at the application level, as part of the process described in the

BPEL specification, and thus the method cannot rely on its existence. It is possible that additional inter-agent communication could prevent these failures from escaping the agents' notice, or that the detection of such failures is unnecessary, however, such questions were not explored as part of this work.

In the cases where a communication failure is detected and has been resolved, it is possible that the agents will no longer be able to engage in the conversation that they previously shared. This may be due to increased restrictions, additional dependencies or reductions in the provided information values. As a result, the agent's workflow script composition algorithm may be unable to construct a complete script that satisfies the provided constraints. The execution of the workflow will thus fail once more, in an unrecoverable fashion. The method's advocacy of modular BPEL specifications may facilitate developer-assisted resolution; increased use of semantic web service methods could also be used to address this issue. The composition algorithm could be enhanced to access a repository of available workflow scripts, and could construct data mediators to reconcile data format discrepancies. This would increase the variety of possible composed workflow scripts available to the agent. Such enhancements would suffer from the same deficiency, however, requiring human intervention in some cases.

An important feature of BPEL and its peers is its support for the signalling and recovery of application-level exceptions. While exception signalling between endpoints takes place within the application-defined message exchanges, and thus requires no special treatment within the method's definition, exception signalling and recovery within a process specified in BPEL must be defined in the case of composed workflow scripts. Specifically, the behaviour of the BPEL *throw* activity, and the semantics of fault and compensation handlers within a composed workflow script must be specified as stringently as in the BPEL standard itself. This is necessary to ensure that the composition mechanism enhances, but does not restrict the possible uses of BPEL specifications.

5.2.2. Implementation Limitations

The implementation of the WRABBIT system is in many ways incomplete. Nowhere is this more obvious, however, than with its support of the BPEL specification. The current implementation supports only the *receive*, *reply*, *invoke*, *assign*, and *sequence* activities.

Because this list does not include any looping or conditional execution constructs, many workflow scripts that can be modelled in BPEL cannot yet be executed by a WRABBIT agent. Therefore, the case studies in this work are limited to sequences of message exchanges. Before new case studies are introduced to examine the system's behaviour when executing more complex workflow scripts, this implementation deficiency must be addressed.

Much as the theory presented in this paper does not address application-level failures, the implementation provides no support for them. Even when executing non-abstract, uncomposed workflow scripts that are unaffected by the missing theory identified in the previous section, the implementation does not support the *throw* activity, or fault and compensation handlers. Further, the current ACL used by the WRABBIT system does not support the transmission of WSDL fault messages, and no mechanism for distinguishing these from the expected response message has been selected (possibilities include a new performative or message field). In order to examine the interaction of application-level and conversation-level errors, this support must be added to the system's implementation.

Bibliography

[ACKM04] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services: Concepts, Architectures and Applications*. Springer-Verlag. 2004.

[AVMM04] Rohit Aggarwal, Kunal Verma, John Miller, and William Milnor. "Constraint Driven Web Service Composition in METEOR-S". 206-219. *Proceedings of IEEE International Conference on Services Computing*. June 2004.

[BDFRS02] B. Benatallah, M. Dumas, M. Fauvet, F. A. Rabhi, and Quan Z. Sheng. "Overview of some patterns for architecting and managing composite web services". 9-16. *SIGecom Exchanges*. 3. 3. June 2002.

[BDFR02] B. Benatallah, M. Dumas, M. Fauvet, and F. A. Rabhi. "Towards Patterns of Web Services Composition". 301-313. *Patterns and Skeletons for Parallel and Distributed Computing*. S. Gorlatch and F. Rabhi. Springer-Verlag. 2002.

[BF95] Mihai Barbuceanu and Mark S. Fox. "COOL: A language for describing coordination in multi agent systems". 17-24. *Proceedings of the International Conference on Multiagent Systems (ICMAS-95)*. June 2002.

[BIP88] Michael E. Bratman, David J. Israel, and Martha E. Pollack. "Plans and resource-bounded practical reasoning". 349-355. *Computational Intelligence*. 4. 4. 1988.

[Bla04] M. Brian Blake. "Forming Agents for Business Process Orchestration". *Proceedings of the 37th Hawaii International Conference on System Sciences (HICSS-37), Web Services and Workflow Track*. January 2004.

[Bla02] M. Brian Blake. "An Agent-Based Cross-Organizational Workflow Architecture in Support of Web Services". *Proceedings of the 11th IEEE Workshop on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE 2002)*. June 2002.

[BPEL4WS] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. *Business Process Execution Language for Web Services (BPEL4WS) Version 1.1*. BEA Systems, IBM Corporation, Microsoft Corporation, SAP AG, Siebel Systems. May, 2003.

[BV05] Paul Buhler and José Vidal. "Towards Adaptive Workflow Enactment Using Multiagent Systems". 61-87. *Information Technology and Management Journal.* 6. 1. 2005.

[BV04] Paul Buhler and José Vidal. "Enacting BPEL4WS Specified Workflows with Multiagent Systems". *Proceedings of the Workshop on Web Services and Agent-Based Engineering.* 2004.

[CCMN04] Girish Chafle, Sunil Chandra, Vijay Mann, and Mangala G. Nanda. "Decentralized Orchestration of Composite Web Services". *Proceedings of the Alternate Track on Web Services at the 13th International World Wide Web Conference (WWW 2004).* May 2004.

[CF04] Carine Courbis and Anthony Finkelstein. "Towards an Aspect Weaving BPEL Engine". *Proceedings of the Third AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS'04).* 2004.

[CM05] Emilia Cimpian and Adrian Mocan. *Process Mediation in WSMX Version 0.1.* Digital Enterprise Research Institute (DERI). July, 2005.

[DL95] Keith S. Decker and Victor R. Lesser. "Designing a family of coordination algorithms". 64-84. *Proceedings of the Thirteenth International Workshop on Distributed Artificial Intelligence.* 1995.

[DL93] Keith S. Decker and Victor R. Lesser. "Quantitative modelling of complex computational task environments". 217-224. *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI 93).* 1993.

[EP05] Renée Elio and Anita Petrinjak. "Normative communication models for agent error messages". 273-305. *Autonomous Agents and Multi-Agent Systems.* 11. 3. Springer-Verlag. 2005.

[EPTS01] Gregg Economou, Massimo Paolucci, Maksim Tsvetovat, and Katia Sycara. "Interaction without commitments: An initial approach". *Agents 2001.* 2001.

[FD05] Cristina Feier and John Domingue. *WSMO Primer Version 0.1.* Digital Enterprise Research Institute (DERI). April, 2005.

[FIPAActs] Foundation for Intelligent Physical Agents. *FIPA Communicative Act Library Specification.* Foundation for Intelligent Physical Agents. 2002.

[FIPAStruct] Foundation for Intelligent Physical Agents. *FIPA ACL Message Structure*

Specification. Foundation for Intelligent Physical Agents. 2002.

[FBS04] Xiang Fu, Tevfik Bultan, and Jianwen Su. "Analysis of interacting BPEL web services". 621-630. *Proceedings of the 13th international Conference on World Wide Web (WWW 2004)*. ACM Press. 2004.

[FUKM04] Howard Foster, Sebastian Uchitel, Jeff Kramer, and Jeff Magee. "Compatibility Verification for Web Service Choreography". *Proceedings of the second IEEE International Conference on Web Services (ICWS 2004)*. July, 2004.

[FUMK05] Howard Foster, Sebastian Uchitel, Jeff Magee, and Jeff Kramer. "Tool Support for Model-Based Engineering of Web Service Compositions". *Proceedings of the third IEEE International Conference on Web Services (ICWS 2005)*. July, 2005.

[GHB00] Mark Greaves, Heather Holmback, and Jeffrey Bradshaw. "What is a conversation policy?". 118-131. *Issues in Agent Communication (LNAI 1916)*. Frank Dignum and Mark Greaves. Springer-Verlag. 2000.

[GPPTW88] Michael Georgeff, Barny Pell, Martha Pollack, Milind Tambe, and Michael Wooldridge. "The Belief-Desire-Intention model of agency". 349-355. *Proceedings of Agents, Theories, Architectures, and Languages*. 1999.

[HCMOB05] Armin Haller, Emilia Cimpian, Adrian Mocan, Eyal Oren, and Christoph Bussler. "WSMX - A Semantic Service-Oriented Architecture". *Proceedings of the third IEEE International Conference on Web Services (ICWS 2005)*. July, 2005.

[KD95] Mark Klein and Chrysanthos Dellarocas. "Exception handling in agent systems". 62-68. *Proceedings of the Third International Conference on Autonomous Agents (Agents '99)*. J. M. Bradshaw, O. Etzioni, and J. Mueller. ACM Press. 1999.

[KGR96] David Kinny, Michael Georgeff, and Anand Rao. "A Methodology and Modelling Technique for Systems of BDI Agents". *Agents Braking Away, Seventh European Workshop on Medelling Autonomous Agents in a Multi-Agent World (MAAMAW 96)*. 1996.

[Kre01] Heather Kreger. *Web Services Conceptual Architecture (WSCA 1.0)*. IBM Software Group. 2001.

[LF97] Yannis Labrou and Tim Finin. "A proposal for a new KQML specification". *Technical Report #CS-97-03*. Computer Science and Electrical Engineering

Department, University of Maryland. 1997.

[Moo00] Scott Moore. “On conversation policies and the need for exceptions”. 144-159. *Issues in Agent Communication (LNAI 1916).* Frank Dignum and Mark Greaves. Springer-Verlag. 2000.

[MC05] Adrian Mocan and Emilia Cimpian. *WSMX Mediation Version 0.1.* Digital Enterprise Research Institute (DERI). September, 2004.

[MM03] Daniel Mandell and Sheila McIlraith. “Adapting BPEL4WS for the Semantic Web: The Bottom-Up Approach to Web Service Interoperation”. *The Proceedings of the Second International Semantic Web Conference (ISWC 2003).* 2003.

[MWF05] Bendick Mahleko, Andreas Wombacher, and Peter Fankhauser. “A Grammar-Based Index for Matching Business Processes”. *Proceedings of the third IEEE International Conference on Web Services (ICWS 2005).* July, 2005.

[NK04] Mangala G. Nanda and Neeran M. Karnik. “Synchronization Analysis for Decentralizing Composite Web Services”. 91-119. *International Journal of Cooperative Information Systems.* 13. 1. March 2004.

[NU00] Marian H. Nodine and Amy Unruh. “Constructing robust conversation policies in dynamic agent communities”. 206-219. *Issues in Agent Communication (LNAI 1916).* Frank Dignum and Mark Greaves. Springer-Verlag. 2000.

[Nwa96] Hyacinth S. Nwana. “Software Agents: An Overview”. 205-244. *Knowledge Engineering Review.* 11. 3. Cambridge University Press. 1996.

[OWL-S] David Martin, Mark Burstein, Jerry Hobbs, Ora Lassila, Drew McDermott, Sheila McIlraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry Payne, Evren Sirin, Naveen Srinivasan, and Katia Sycara. *OWL-S: Semantic Markup for Web Services.* OWL-S Coalition. November, 2004.

[Pel03] Chris Peltz. “Web Services Orchestration: a Review of Emerging Technologies, Tools, and Standards”. *Technical Report.* Hewlett Packard, Co.. January 2003.

[Pet01] Charles Petrie. “Agent-based software engineering”. 58-76. *Agent-Oriented Software Engineering: The State of the Art (LNAI 1957).* P. Ciancarini and M. Wooldridge. Springer-Verlag. 2001.

[PKPSS99] Massimo Paolucci, Dirk Kalp, Anandeep S. Pannu, Onn Shehory, and Katia Sycara. “A Planning Component for RETSINA Agents”. *Lecture Notes in Artificial*

Intelligence, Intelligent Agents VI. Springer-Verlag. 1999.

[POSV04] Abhijit A. Patil, Swapna A. Oundhakar, Amit P. Sheth, and Kunal Verma. "Meteor-s web service annotation framework". 553-562. *Proceedings of the 13th international conference on the World Wide Web.* ACM Press. 2004.

[PR90] Amir Pnueli and Roni Rosner. "Distributed Reactive Systems are Hard to Synthesize". 746-757. *Proceedings of the 31st IEEE Symposium on Foundations of Computer Science.* 1990.

[PTBM05] M. Pistore, P. Traverso, P. Bertoli, and A. Marconi. "Automated Synthesis of Composite BPEL4WS Web Services". *Proceedings of the third IEEE International Conference on Web Services (ICWS 2005).* July, 2005.

[RG95] Anand S. Rao and Michael P. Georgeff. "BDI agents: From theory to practice". 349-355. *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95).* V. Lesser. MIT Press. 1995.

[SOAP] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. *Simple Object Access Protocol (SOAP) 1.1.* World Wide Web Consortium. May, 2000.

[SPVG01] Katia Sycara, Massimo Paolucci, Martin Van Velsen, and Joseph Andrew Giampapa. "The RETSINA MAS Infrastructure". *Technical Report CMU-RI-TR-01-05.* Robotics Institute, Carnegie Mellon University. March, 2001.

[VAGDL04] Kunal Verma, Rama Akkiraju, Richard Goodwin, Prashant Doshi, and Juhnyoung Lee. "On Accommodating Inter Service Dependencies in Web Process Flow Composition". *AAAI Spring Symposium 2004.* 2004.

[WBLX00] Thomas Wagner, Brett Benyo, Victor Lesser, and Ping Xuan. "Investigating interactions between agent conversations and agent control components". 314-331. *Issues in Agent Communication (LNAI 1916).* Frank Dignum and Mark Greaves. Springer-Verlag. 2000.

[WS-AtomicTransaction] Luis Cabrera, George Copeland, Max Feingold, Tom Freund, Jim Johnson, Chris Kaler, Johannes Klein, David Langworthy, Anthony Nadalin, David Orchard, Ian Robinson, Tony Storey, and Satish Thatte. *Web Services Atomic Transaction (WS-AtomicTransaction).* BEA Systems, International Business Machines Corporation, Microsoft Corporation. November, 2004.

[WS-BusinessActivity] Luis Cabrera, George Copeland, Tom Freund, Johannes Klein, David Langworthy, Frank Leymann, David Orchard, Ian Robinson, Tony Storey, and Satish Thatte. *Web Services Business Activity Framework (WS-BusinessActivity).* BEA Systems Inc., IBM Corporation, Microsoft Corporation. November, 2004.

[WS-CDL] Nickolas Kavantzas, David Burdett, Gregory Ritzinger, Tony Fletcher, and Yves Lafon. *Web Services Choreography Description Language Version 1.0 (WS-CDL) - Working Draft.* World Wide Web Consortium W3C. 17 December 2004.

[WS-Coordination] Luis Cabrera, George Copeland, Max Feingold, Tom Freund, Jim Johnson, Chris Kaler, Johannes Klein, David Langworthy, Anthony Nadalin, David Orchard, Ian Robinson, John Shewchuk, and Tony Storey. *Web Services Coordination (WS-Coordination).* BEA Systems, International Business Machines Corporation, Microsoft Corporation. November, 2004.

[WS-Policy] Don Box, Francisco Curbera, Maryann Hondo, Chris Kaler, Dave Langworthy, Anthony Nadalin, Nataraj Nagaratnam, Mark Nottingham, Claus von Riegen, and John Shewchuk. *Web Services Policy Framework (WS-Policy).* BEA Systems, International Business Machines Corporation, Microsoft Corporation, SAP AG. June, 2003.

[WS-PolicyAttachment] Don Box, Francisco Curbera, Maryann Hondo, Chris Kaler, Hiroshi Maruyama, Anthony Nadalin, David Orchard, Claus von Riegen, and John Shewchuk. *Web Services Policy Attachment (WS-PolicyAttachment).* BEA Systems, International Business Machines Corporation, Microsoft Corporation, SAP AG. June, 2003.

[WS-SecurityPolicy] Giovanni Della-Libera, Martin Gudgin, Phillip Hallam-Baker, Maryann Hondo, Hans Granqvist, Chris Kaler, Hiroshi Maruyama, Michael McIntosh, Anthony Nadalin, Nataraj Nagaratnam, Rob Philpott, Hemma Prafullchandra, John Shewchuk, Doug Walter, and Riaz Zolfonoon. *Web Services Security Policy Language (WS-SecurityPolicy).* International Business Machines Corporation, Microsoft Corporation, RSA Security Inc., and VeriSign Inc.. July, 2005.

[WSDL] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. *Web Services Description Language (WSDL) 1.1*. World Wide Web Consortium. March, 2001.

[WS-IBasicProfile] Keith Ballinger, David Ehnebuske, Christopher Ferris, Martin Gudgin, Canyang Kevin Liu, Mark Nottingham, and Prasad Yendluri. *Basic Profile Version 1.1*. Web Services Interoperability Organization. 2004.