D2.4.14 Decentralized Orchestration of Composite Services

Coordinator Walter Binder (EPFL)

Radu Jurca (EPFL),
Boi Faltings (EPFL)
Ian Blacoe (UniLiv)
Valentina Tamma (UniLiv)
Mike Wooldridge (UniLiv)

Abstract.
EU-IST Network of Excellence (NoE) IST-2004-507482 KWEB
Deliverable D2.4.14 (WP2.4)

This deliverable focusses on two mechanisms for integrating independent services. It is divided into two chapters, Chapter 2 presents a flexible approach for efficiently executing composite services in a decentralized way, whilst Chapter 3 presents a semantic based approach to coordination of services.

Keyword list: Composite Services, Workflows, Decentralized Orchestration, Coodination, Web Services

Document Identifier	KWEB/2007/D2.4.14
Project	KWEB EU-IST-2004-507482
Version	v1.1
Date	February 02, 2008
State	Final
Distribution	public

Copyright © 2008 The contributors

Knowledge Web Consortium

This document is part of a research project funded by the IST Programme of the Commission of the European Communities as project number IST-2004-507482.

University of Innsbruck (UIBK) - Coordinator
Institute of Computer Science
Technikerstrasse 13
A-6020 Innsbruck
Austria
Contact person: Dieter Fensel
E-mail address: dieter.fensel@uibk.ac.at

École Polytechnique Fédérale de Lausanne (EPFL)
Computer Science Department
Swiss Federal Institute of Technology
IN (Ecublens), CH-1015 Lausanne
Switzerland
Contact person: Boi Faltings
E-mail address: boi.faltings@epfl.ch

France Telecom (FT)
4 Rue du Clos Courtel
35512 Cesson Sévigné
France. PO Box 91226
Contact person : Alain Leger
E-mail address: alain.leger@rd.francetelecom.com

Freie Universität Berlin (FU Berlin)
Takustrasse 9
14195 Berlin
Germany
Contact person: Robert Tolksdorf
E-mail address: tolk@inf.fu-berlin.de

Free University of Bozen-Bolzano (FUB)
Piazza Domenicani 3
39100 Bolzano
Italy
Contact person: Enrico Franconi
E-mail address: franconi@inf.unibz.it

Institut National de Recherche en Informatique et en Automatique (INRIA)
ZIRST - 655 avenue de l'Europe -
Montbonnot Saint Martin
38334 Saint-Ismier
France
Contact person: Jérôme Euzenat
E-mail address: Jerome.Euzenat@inrialpes.fr

Centre for Research and Technology Hellas / Informatics and Telematics Institute (ITI-CERTH)
1st km Thermi - Panorama road
57001 Thermi-Thessaloniki
Greece. Po Box 361
Contact person: Michael G. Strintzis
E-mail address: strintzi@iti.gr

Learning Lab Lower Saxony (L3S)
Expo Plaza 1
30539 Hannover
Germany
Contact person: Wolfgang Nejdl
E-mail address: nejdl@learninglab.de

National University of Ireland Galway (NUIG)
National University of Ireland
Science and Technology Building
University Road
Galway
Ireland
Contact person: Christoph Bussler
E-mail address: chris.bussler@deri.ie

The Open University (OU)
Knowledge Media Institute
The Open University
Milton Keynes, MK7 6AA
United Kingdom
Contact person: Enrico Motta
E-mail address: e.motta@open.ac.uk

Universidad Politécnica de Madrid (UPM)
Campus de Montegancedo sn
28660 Boadilla del Monte
Spain
Contact person: Asunción Gómez Pérez
E-mail address: asun@fi.upm.es

University of Karlsruhe (UKARL)
Institut für Angewandte Informatik und Formale
Beschreibungsverfahren - AIFB
Universität Karlsruhe
D-76128 Karlsruhe
Germany
Contact person: Rudi Studer
E-mail address: studer@aifb.uni-karlsruhe.de

University of Liverpool (UniLiv)
Chadwick Building, Peach Street
L697ZF Liverpool
United Kingdom
Contact person: Michael Wooldridge
E-mail address: M.J.Wooldridge@csc.liv.ac.uk

University of Manchester (UoM)
Room 2.32. Kilburn Building, Department of Computer
Science, University of Manchester, Oxford Road
Manchester, M13 9PL
United Kingdom
Contact person: Carole Goble
E-mail address: carole@cs.man.ac.uk

University of Sheffield (USFD)
Regent Court, 211 Portobello street
S14DP Sheffield
United Kingdom
Contact person: Hamish Cunningham
E-mail address: hamish@dcs.shef.ac.uk

University of Trento (UniTn)
Via Sommarive 14
38050 Trento
Italy
Contact person: Fausto Giunchiglia
E-mail address: fausto@dit.unitn.it

Vrije Universiteit Amsterdam (VUA)
De Boelelaan 1081a
1081HV. Amsterdam
The Netherlands
Contact person: Frank van Harmelen
E-mail address: Frank.van.Harmelen@cs.vu.nl

Vrije Universiteit Brussel (VUB)
Pleinlaan 2, Building G10
1050 Brussels
Belgium
Contact person: Robert Meersman
E-mail address: robert.meersman@vub.ac.be

Work package participants

The following partners have taken an active part in the work leading to the elaboration of this document:

Ecole Polytechnique Fédérale de Lausanne
University of Liverpool

Changes

Version	Date	Author	Changes
0.1	20.01.07	Walter Binder	Creation
0.2	18.02.07	Radu Jurca	Model of service invocation triggers
0.3	12.04.07	Walter Binder	API
0.4	22.06.07	Radu Jurca	Evaluation seetings
0.5	10.07.07	Radu Jurca	Evaluation results
0.6	19.09.07	Walter Binder	Figure and example
0.7	15.10.07	Walter Binder	Security, failure handling
0.8	25.11.07	Walter Binder	Introduction, related work, conclusion
0.9	15.12.07	Valentina Tamma	Addition of the coordination, update of introduction and conclusions
1.0	09.01.08	Walter Binder	Minor updates of all sections
1.1	02.02.08	Walter Binder	Addressed reviewer comments

Executive Summary

Service-oriented computing, a new approach to software development, enables the construction of distributed applications by integrating services that are available over the web [31]. The building blocks of such applications are web services[1] that are accessed using standard protocols. In addition, several mechanisms contribute to the integration and seamless execution of services independently developed. In this deliverable we focus our attention on two of these mechanisms, namely the composition of individual web services into a new composit one, and the coordination of the execution of independent services that make use of scarce resources.

Traditional, centralized workflow orchestration often leads to inefficient routing of messages. To solve this problem, we present a novel scheme to execute workflows in a fully decentralized way. We introduce service invocation triggers, a lightweight infrastructure that routes messages directly from the producing service to the consuming one, enabling fully decentralized workflow orchestration. An evaluation confirms that decentralized orchestration can significantly reduce the network traffic when compared with centralized orchestration.

The coordination of independent services whose execution requires some *coordination* mechanisms in order to avoid possible conflicts when accessing resources. Indeed, coordination is a fundamental problem in open distributed environments where multiple independent parties require access to shared resources that might be scarce, or be shared to a different extent. In this deliverable we present a run-time mechanism for coordinating the execution of independent services accessing scarce resourse. This mechanism is based on an ontological description of the notion of conflict and of the rules to achieve coordination. The mechanisms has been implemented as a web service, and an initial evaluation is presented here.

[1] We use the terms *web service* and *service* interchangeably.

Contents

Chapter 1

Introduction

Service-oriented computing, a new approach to software development, enables the construction of distributed applications by integrating services that are available over the web [31]. The building blocks of such applications are web services[1] that are accessed using standard protocols. In addition, several mechanisms contribute to the integration and seamless execution of services independently developed. In this deliverable we focus our attention on two of these mechanisms, namely the composition of individual web services into a new composit one, and the coordination of the execution of independent services that make use of scarce resources.

The composition of individual web services into an added-value, composite web service is usually represented as a workflow. In previous work [17, 15] we presented a flexible and efficient framework for the fully automated generation of composite web services based on a given service request and a potentially large-scale repository of web service advertisements. In this deliverable we complement our service composition infrastructure with a mechanism for the efficient, distributed execution of composite web services represented as workflows.

Even though a workflow may invoke services distributed over multiple servers, the orchestration of the workflow is typically centralized. E.g., BPWS4J [11] acts as centralized coordinator for all interactions among the individual services within a workflow. While this approach gives complete control over the workflow orchestration to a single entity (which may monitor the progress), it often leads to inefficient communication, as all intermediary results are transmitted to the central workflow orchestration site, which may become a bottleneck. This is particularly problematic, if a workflow is executed on a mobile device with limited or expensive network connection.

The first contribution of this deliverable is a novel scheme of fully decentralized workflow orchestration. We introduce *service invocation triggers*, in short *triggers*, which act as proxies for individual service invocations. Triggers collect the required input data

[1] We use the terms *web service* and *service* interchangeably.

before they invoke the service, i.e., triggers are also buffers. Moreover, they forward service outputs to exactly those sites where they are actually needed, supporting multicast. In order to make use of triggers, workflows are decomposed into sequential fragments (dataflows), which contain neither loops nor conditionals, and the data dependencies within each workflow fragment are encoded within the triggers. In this deliverable we focus on the orchestration of sequential workflow fragments; in the following, the term *workflow* stands for a sequential workflow fragment. Once the trigger of the first service in a workflow has received all input data, the execution of that service is started and the outputs are forwarded to the triggers of subsequent services. Consequently, the workflow is executed in a fully decentralized way, the data is transmitted directly from the producer to all consumers.

For the discussion in this deliverable, a simplified formalism to describe services is sufficient. We describe a service by a set of input and a set of output parameters. Each input (resp. output) parameter has an associated name that is unique with the set of input (resp. output) parameters. We assume the workflow of a composite service to be consistent with the specifications of the individual services. Hence, we do not consider the type of parameters, as we presume that whenever a service receives an input for a particular parameter, the actual type of the passed value corresponds to the formal type of that service parameter.

We assume that services are invoked by remote procedure calls (RPC), such as SOAP RPC [43]. The values for the input parameters are provided in a request message, while the values for the output parameters are returned in a response message. Asynchronous (one-way) calls can be easily mapped to RPC, which is actually the case for SOAP over HTTP. Our triggers are designed to be transparent to services, i.e., services do not need to know whether they are invoked directly by a client or by a trigger. Therefore, our framework can be deployed without changing existing services.

The second contribution of this deliverable focusses on the coordination of independent services whose execution requires some *coordination* mechanisms in order to avoid possible conflicts when accessing resources. Indeed, coordination is a fundamental problem in open distributed environments where multiple independent parties require access to shared resources that might be scarce, or be shared to a different extent. If two services wish to use the same resource simultaneously then their activities are likely to affect one another. If the resource is nonshareable then this will naturally lead to problems - a lost update perhaps, but in the worst case even damage to the resource. In other words the two activities do not act in isolation; there exists an interdependency between them. In another situation, two activities may use different resources, but one of the activities may require the results of the other activity before it can begin. Again, an interdependency exists between the activities. Such interdependencies need not be destructive, however. In fact, they can be beneficial. For example, suppose two processes are carrying out separate tasks, both of which rely on some intermediate computation. In this situation it makes sense for the two processes to proactively exchange information so that effort is not duplicated. Here, an interdependency exists between the two activities which, if exploited

correctly, can increase the overall utility of the system. However, failure to exploit the interdependency does not prevent the processes from successfully completing their tasks.

Coordination then can be described as the management of interdependencies amongst activities [27]. It has been the subject of extensive research, not least in the field of multi-agent systems [33]. One approach for providing coordination is to use hard-wired low-level constructs such as semaphores or locks to ensure that various activities do not destructively interfere with one another [4]. This method, known as *synchronisation*, is useful in static environments, where the resources and activities comprising the system are well known in advance and can be taken into account at design-time. However, in more open systems in which resources can come and go and may constantly evolve it is often impossible to anticipate every eventuality at design-time and this approach fails.

In such systems, we ideally want to provide services or agents with the ability to reason about activities they wish to perform and the coordination issues that will arise, thereby enabling them to resolve any interdependencies *autonomously* [19]. To achieve this, the services will need to communicate with one another about their intentions to utilise particular resources. This communication will in turn require an agreed common vocabulary with explicit semantics so that all the services can communicate in the same terms. In other words the services will require an *ontology* of coordination. This paper details such an ontology [38] based on previous work by the multi-agents system community [19, 27, 34, 42]. It also details a set of rules which can be used to manage activities and resolve any interdependencies that may exist between them [29].

To determine the feasibility of this approach, a web service was developed containing the ontology and rules. This service acts as a centralised coordinator with which resources can be registered. Agents or services[2] can then request to perform activities using these resources and the service will coordinate the various requests, detecting and resolving interdependencies appropriately.

To accompany the web service we have implemented a visualisation client, which enables us to submit the various calls to the service. The client monitors the internal state of the service and uses a Gantt chart to illustrate the various resources, activities and interdependencies to the user.

The rest of this deliverable is composed of mainly two parts, Chapter 2 looks at the orchestration of individual services, and it is organised as follows: Section 2.1 introduces the concept of triggers, while Section 2.2 presents a simple API to create and manipulate triggers. Section 2.3 explains how workflows are mapped to triggers in order to execute them in a fully decentralized way, which is illustrated with an example. In Section 2.4 we consider the handling of failures during workflow execution. Section 2.5 treats security aspects related to the use of triggers. Section 2.6 presents an evaluation confirming that decentralized orchestration allows to significantly reduce network traffic in comparison with centralized orchestration, while Section 2.7 discusses related work.

[2]In the reminder of the deliverable we will use the terms services and agents interchangeably

The aim of Chapter 3 is to illustrate the coordination ontology and rules, provide a brief overview of the coordination service, and describe how this system can be applied to a real-life use case taken from the domain of car insurance. An overview of the background work in coordination is given in Section 3.1. Section 3.2 describes how this is translated into the coordination ontology. The set of rules that implement the coordination mechanism are then presented in Section 3.3 before a description of the coordination service implementation and visualisation client are given in Section 3.4. Section 3.5 then demonstrates how the approach can be applied to a use case. Finally we draw conclusions in Chapter 4.

Chapter 2

Workflow Orchestration

2.1 Service Invocation Triggers

In this chapter we give an overview of service invocation triggers, the main abstraction in our framework for efficient, decentralized workflow orchestration. A trigger corresponds to one invocation of a service. Hence, it can be considered a specialized proxy for a single service invocation. A trigger plays four different roles:

1. It collects the input parameter values for one service invocation.
2. It acts as a message buffer, as each input parameter value may be transmitted by a distinct sender at a different time.
3. It triggers the service invocation, after a value has been received for each required input parameter (synchronization). In order to invoke the service, the trigger assembles a RPC request message.
4. It defines the routing for each output parameter value of the service. As each output parameter value may be routed differently, the trigger may have to split the RPC response message returned by the service upon invocation. Each output parameter value may be routed to multiple different triggers, i.e., triggers support multicast.

With the aid of triggers it is possible to distribute the knowledge concerning the data dependencies of the services within a workflow. The main difference between the workflow and the corresponding triggers is that the triggers are distributed and attached to services. Each trigger defines which service to invoke. The trigger waits until all required input parameter values are available before it fires (i.e., triggers the service invocation).

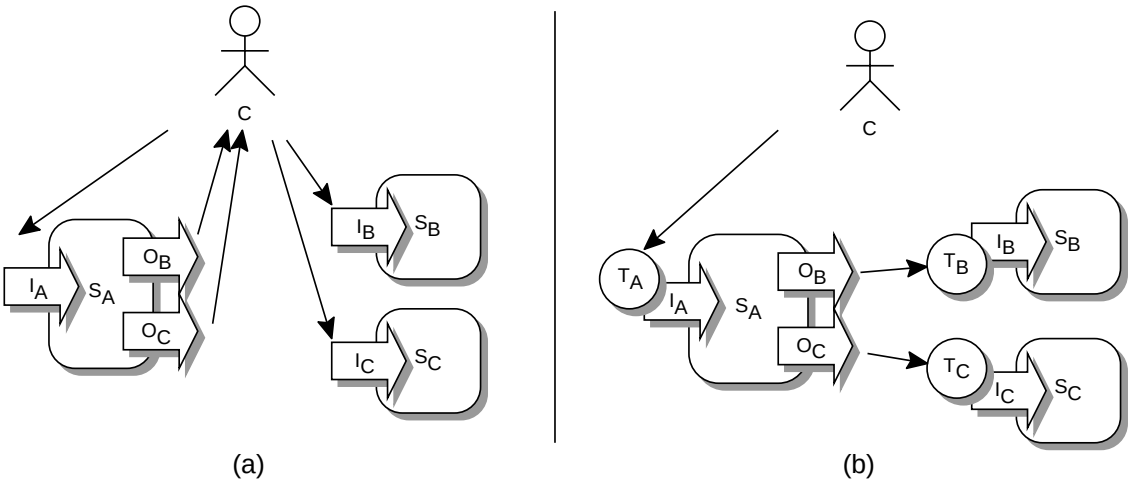

Figure 2.1: (a) Centralized orchestration versus (b) decentralized orchestration using triggers.

Moreover, each trigger encapsulates workflow-specific knowledge where the output parameter values of the service invocation are needed. As the trigger acts as a proxy for the service, it receives the output parameter values and forwards them to other triggers according to its routing information.

The following example (see Figure 2.1) shows how triggers can help to optimize the data transmission between services: Consider a service S_A which requires a single input parameter I_A and produces two output parameters O_B and O_C, while the services S_B and S_C both require a single input parameter, I_B resp. I_C. Further, assume a workflow fragment that requires an invocation of S_A with a given value of I_A, passing the value of the output parameter O_B to service S_B (as input parameter I_B) and O_C to service S_C (as input parameter I_C). If the workflow orchestration is managed in a centralized way by the client C (e.g., C may represent a centralized workflow orchestration engine), C has to send I_A to S_A, S_A will return the output parameter values O_B and O_C to C, which C will pass on to S_B and S_C (see Figure 2.1 (a)). That is, the output parameter values O_B and O_C are not directly sent to the place where they are needed, but they are routed through C. This routing may not be optimal, if we assume that C is not interested in O_B and O_C (which may represent intermediary results of a composite service). Considering that C may be connected to a slow or expensive network (e.g., C may use a mobile device with a wireless network connection), the negative impacts of the centralized workflow orchestration become immediately apparent.

In order to optimize the data transmission between services, a trigger shall be installed as closely as possible to the service it will invoke. In the example before, C may create the triggers T_A, T_B, and T_C for the invocation of the services S_A, S_B, and S_C. T_A shall be close to S_A, T_B to S_B, and T_C to S_C. As triggers allow to define the routing of output parameter values, T_A can be configured to directly send O_B to T_B and O_C to T_C, avoiding

to pass these intermediary results through C (see Figure 2.1 (b)).

Even though it is advantageous to install triggers close to the actual services they are triggering, they may be set up on an arbitrary site. For instance, if the provider of S_B supports triggers, he may accept triggers directly in the same server running service S_B, which will result in efficient local communication between T_B and S_B. If it is not possible to colocate triggers and services within the same server, the provider of S_B may offer a separate server dedicated to triggers in his local network. Otherwise, dedicated servers may offer to host arbitrary triggers.

Using triggers, many different workflow orchestration schemes can be implemented. If all triggers are hosted by the client, it corresponds to the centralized workflow orchestration model described before. If all triggers are hosted by a dedicated server, it corresponds to passing the workflow to a dedicated orchestration engine which executes it in a centralized way. If each trigger is installed locally with the service that it will invoke, the workflow is executed in a fully decentralized way, delivering intermediary results only to those places where they are needed. Our framework does not dictate any of these settings. Therefore, it is possible to bootstrap the support for triggers by deploying dedicated servers to host triggers. With the time, service providers may start to directly support triggers in their environments (incremental deployment).

2.2 Defining Triggers

In this chapter we present more details regarding the installation of triggers. In our description we use the following abbreviations for identifying services respectively triggers:

SID: Service ID. Globally unique identifier of a service to be invoked. It consists of host, port, protocol, and local service identifier (e.g., service name and version number, depending on the protocol). SIDs can be computed from service descriptions, including grounding information.

PID: Parameter ID. Locally unique identifier for a service input or output parameter.

TID: Trigger ID. Globally unique trigger identifier. It consists of host, port, and a local trigger identifier (e.g., an integer number referring to a trigger).

In the following we present a simple API to deal with triggers in an abstract way:

CreateTrigger: Creates and installs a trigger.

Arguments:

- Destination of the trigger (host and port). `CreateTrigger` will ask the destination to set up the desired trigger.

- SID. The service to be invoked by the trigger.
- Service input parameters to wait for. Each parameter is identified by its PID. A parameter may be required or optional. The trigger will fire as soon as all required input parameters are available. As for a given input parameter multiple values may arrive before the trigger fires (while still some of the required input parameters are missing), the client has to define which values to preserve: `preserveLast` or `preserveFirst`. If values for optional input parameters arrive before the trigger fires, they will be passed to the service. After the trigger has fired, arriving input parameter values are discarded.
- Optional: Input data. For each input parameter, a default value may be provided. This value could be transmitted with `SendData` (see below), but including it in `CreateTrigger` may be more efficient and help to reduce network traffic.
- Output routing. For each output parameter (identified by a PID_O) generated by the service SID, the output routing defines a possibly empty list of pairs (TID_i, PID_i) to forward the output parameter value. I.e., whenever the service SID returns a value for the output parameter PID_O, the trigger will forward it to all triggers TID_i as input parameter PID_i, implementing a multicast. If there is a communication problem with a trigger TID_i, the trigger will retry to forward the data several times in order to overcome temporary network problems.
- Desired timeouts:
  1. Timeout to wait for input parameter values, starting with the installation of the trigger. If not all required input parameter values arrive before this timeout, the trigger will be discarded.
  2. Timeout to wait for service completion, starting with the service invocation.
  3. Timeout to wait for completed forwarding of output parameter values, starting when the trigger receives the response of the service invocation.
- Optional: Destination for failure notification message (host, port, protocol). In the case of a failure (i.e., service returning a failure message or expiration of one of the timeouts mentioned before), a failure notification is sent before the trigger is discarded, including information concerning the current state of the trigger. The level of detail of this notification can be configured. The message may simply indicate the reason of the failure, or it may include input resp. output parameter values the trigger has received so far. This information may help the client to recover from the failure.

Results:

- TID of the installed trigger (if the trigger was accepted).

- Granted timeouts. Each granted timeout may be the desired timeout or shorter.

RemoveTrigger: Explicitly removes a trigger. Normally, a trigger is removed automatically if either a timeout occurs or if the output routing task is completed, i.e., all output parameter values have been forwarded according to the trigger's routing information.

Arguments:

- TID. The trigger to remove.

SendData: Sends input parameter values to a trigger. Normally, triggers receive input parameter values either by initialization (see the input data of `CreateTrigger`) or through other triggers (forwarded output parameter values from other services). However, a client may want to install a trigger and provide input parameter values later on.

Arguments:

- TID. The trigger to send data to.
- Input data. For each input parameter, a value may be provided.

Status: Returns information concerning the status of a trigger. I.e., whether the trigger is still waiting for required input, which input parameters have been received so far, whether it has already triggered the service, whether it is waiting for the service output parameters, etc. `Status` enables monitoring the progress of workflow execution.

Arguments:

- TID. The trigger to ask for its status.

Results:

- Status information.

Three different protocols are involved in the communication with triggers and services:

1. **Trigger–service:** The trigger communicates with the service using remote procedure calls (e.g., SOAP RPC [43]). I.e., the trigger is transparent to the service, it behaves as any other client.

2. **Trigger–trigger:** The communication between triggers is unidirectional. A trigger forwards results to other triggers. The message sent from trigger T_A to trigger T_B contains at least one value for an input parameter T_B is waiting for. Even though the communication protocol between triggers need not necessarily comply with

standards, SOAP messages are well suited for trigger–trigger communication. If the service invocation has failed, the trigger does not send any message on the normal output routing path, but it may generate a failure notification message (if specified in `CreateTrigger`). Subsequent triggers will notice the failure by a timeout.

3. **Client–trigger:** A dedicated, simple protocol supports the API primitives described before. For instance, `CreateTrigger` will try to set up a trigger on the specified destination platform.

In the following we present a few aspects of our trigger implementation in Java. A discussion of implementation details had to be omitted due to space limitations. Our system is based on Axis [3], an implementation of SOAP [43]. There are four different service styles in Axis: Three of them ('RPC', 'Document', and 'Wrapped') provide different ways of XML to Java binding. The fourth one, called 'Message', allows to receive and return arbitrary XML data in the SOAP envelope without any type mapping (no data binding). In our system, triggers use the 'Message' style, while we still assume that received messages use the 'RPC' style encoding. In this encoding, a RPC request message is modeled as an outer XML element, which matches the operation name and contains inner XML element tags mapping to service parameters. A trigger extracts all input parameters from incoming messages and merges them into a single new RPC request message to invoke the actual service. The RPC response message is split in order to create outgoing messages for each of the output parameters.

2.3 Decentralized Workflow Orchestration

In this chapter we show how a workflow can be executed using triggers. First, the client decomposes a given workflow into sequential parts that contain neither loops nor conditionals (dataflows). The common workflow patterns [40] *sequence*, *parallel split*, and *synchronization* can be directly expressed as data dependencies between triggers, whereas control structures, such as conditionals or loops, require workflow decomposition. However, considering automatically composed services, this limitation is often not relevant, since most current algorithms for automated service composition create only sequential workflows [17].

Each sequential workflow fragment is executed in the following way:

1. The client creates a (temporary) local service with SID S_{client} to handle the final results of the workflow fragment.

2. The client uses `CreateTrigger` to locally install a trigger to handle the workflow results, referring to S_{client}. TID_{client} is the resulting TID.

3. Starting with the last service in the sequential workflow fragment, a trigger is created for each service invocation. The output parameter values of the last service shall be routed to TID_{client}. The order of setting up the triggers ensures that for each service, the triggers of all subsequent services are created before.

4. Using the optional input data of `CreateTrigger` or `SendData`, the client sends the input parameter values of the workflow fragment to the triggers where these inputs are needed. The trigger of the first service in the workflow fragment will receive all required input parameters and execute the service. The results will be forwarded according to the trigger's output routing, eventually triggering the execuli on of subsequent services. The client is not involved in this process. It is notified of the completed workflow fragment by the invocation of S_{client}.

As an example of the use of triggers, we consider a composite service built from the following simple services:

- `getGPS`: Returns the GPS coordinate of an address.
  Inputs: {`adr_text`}, Outputs: {`adr_gps`}
- `getRoute`: Computes a route between two GPS coordinates as a TIFF image.
  Inputs: {`from_gps`, `to_gps`}, Outputs: {`route_tiff`}
- `tiff2jpeg`: Converts a TIFF image to a JPEG image.
  Inputs: {`image_tiff`}, Outputs: {`image_jpeg`}

We assume that as input the client provides two addresses, `start` and `destination`. The composite service generates a map illustrating the route between these two addresses. As the client has a mobile device with limited memory and a slow wireless network connection, the map shall be delivered in JPEG format with high compression (`result`). The composite service may be described by the following workflow. The variables `tmp1`, `tmp2`, and `tmp3` are intermediary results the client is not interested in.

1. `getGPS`(`adr_text` ← `start`): (`adr_gps` → `tmp1`)
2. `getGPS`(`adr_text` ← `destination`): (`adr_gps` → `tmp2`)
3. `getRoute`(`from_gps` ← `tmp1`, `to_gps` ← `tmp2`):
   (`route_tiff` → `tmp3`)
4. `tiff2jpeg`(`image_tiff` ← `tmp3`): (`image_jpeg` → `result`)

Certainly, the workflow shall not be executed on the mobile device of the client, since this would require transferring the large uncompressed TIFF image (`tmp3`) to and from

the resource-constrained mobile device. However, with the aid of triggers, the workflow can be executed without transferring any intermediary result to the client. The pseudo-code in Figure 2.2 illustrates how the client sets up the decentralized orchestration of the workflow. For the sake of easy readability, details, such as the negotation for timeouts, are left out intentionally.

With `createLocalService` the client simulates a local service that is able to receive the final result delivered by `tiff2jpeg`. From the client's point of view, the delivery of the result is asynchronous, as `tiff2jpeg` is not directly invoked by the client. In this example we assume that it is possible to install the triggers directly within the different server environments. The triggers are created in reverse order of the service invocation sequence in the workflow. The input parameter values for the invocations of `getGPS` are directly passed with the `CreateTrigger` primitive, i.e., the triggers `TID1` and `TID2` will fire immediately after installation. Note that both invocations of `getGPS` (via triggers `TID1` and `TID2`) may be performed in parallel. The output parameter values will be routed to the trigger `TID3`, which will wait until both `from_gps` and `to_gps` are available. Finally, `TID4` will wait for the data forwarded by `TID3` and pass the output parameter value of `tiff2jpeg` to the client (via `TID0`).

2.4 Failure Handling

In our approach, composite web services are executed in a completely decentralized way. Therefore, it is not easily possible to monitor the progress of each service invocation. As the client will only receive the final results of the composite web service, in general it will notice a failure only after a timeout. In this case, the client may restart the execution of the workflow.

If the used web services are not reliable, this approach may result in bad overall performance, since intermediary results may have to be computed multiple times. Hence, the client should make use of the failure notification mechanism in order to collect partial results that had been computed before the failure has happened. Based on the failure notification mechanism, the client could exploit redundant execution plans in order to replace a failed web service. As an alternative (but inefficient) solution, if the decentralized orchestration of a composite web service fails, the client could simply re-execute the workflow in a centralized fashion (fallback solution).

The client may also use the `Status` primitive of triggers in order to monitor the progress of the execution. Note that `Status` will fail if the trigger has already been removed (i.e., after a timeout or after completing its task). However, `Status` creates additional network traffic, therefore an excessive use of this primitive is not consistent with the principal idea of our approach to minimize the network traffic involving the client.

2.5 Security Issues

In this chapter we briefly discuss topics concerning security that arise due to the use of triggers. We distinguish between existing security infrastructure that may hamper the use of triggers and new security threats because of triggers.

As the placement of the triggers affects the communication paths between services and clients, firewalls may prevent the installation of triggers on certain hosts. For instance, if the client C is allowed to communicate with the services S_A and S_B, S_A may not necessarily be able to directly communicate with S_B. Therefore, a trigger installed close to S_A may fail to directly forward intermediary results from S_A to a trigger colocated with S_B.

A related problem concerns authentication. For example, S_B may want to verify that the origin of a service request is the client C. However, as triggers act as proxies that may collect input parameter values from various sources, authentication may fail. This problem could be mitigated by authenticating only the installation of the trigger, even though this does not ensure the same level of security as authenticating that all input data comes from C. Nonetheless, for the composition of information services that are open to the public, the problems concerning firewalls and authentication usually do not crop up.

The trigger infrastructure may be the target of attacks. For instance, if TIDs are not well protected, an attacker may remove a trigger or send fake data, causing the trigger to fire. Because of the forged input data, the triggered service will compute incorrect results. The client may not notice this kind of attack, because once the triggers have been set up, the client does not control the interaction between services and triggers. This problem may be addressed by using TIDs as capabilities, e.g., by chosing a large random number as a part of the TID. Then, if triggers are communicated only between trusted parties across protected (i.e., encrypted) links, forging TIDs will be very difficult.

The placement of triggers is another important issue. In general, triggers should be installed only on trusted sites, i.e., either on the client side, on the site of the service to be invoked, or on the site of a trusted third party. Otherwise, a trigger deployed on an untrusted site may disclose the collected input data and the output data generated by the triggered service, or forge input resp. output data.

Another issue are denial-of-service attacks. An attacker may create a large number of triggers with maximum timeout, he may send large amounts of input data to these triggers while still one required input parameter is missing. Thus, the triggers will have to process and store a significant amount of input data. However, in principle this problem is not much different from traditional denial-of-service attacks against web services. Services may be invoked very frequently and provided with large amounts of data. Such attacks may be mitigated by limiting the number of concurrent connections and limiting the size of message buffers. Similar techniques may be applied to triggers (i.e., limiting the number of concurrent triggers and limiting the buffer size of each trigger). Triggers may even improve load-balancing, as they are installed before the actual web service invocation

happens. I.e., triggers allow the server to plan ahead the expected load in the near future.

As triggers are automatically removed after service invocation, it is not possible to set up cyclic trigger dependencies, which otherwise could be easily abused for denial-of-service attacks.

2.6 Evaluation

In order to evaluate the benefits of our decentralized orchestration scheme, we simulated the network traffic (i.e., the sum of the sizes of all messages sent over the network during the execution of a workflow) caused by centralized orchestration and by decentralized orchestration using triggers.

The evaluation is based on our testbed for service composition [16], which allows to generate random, acyclic workflows, representing composite web services. Each workflow has a random number of nodes N ($3 \leq N \leq 15$). Two of them are special nodes, *START* and *END*, which represent the source of the initial input messages and the destination of the final output messages. All other nodes represent service invocations. Concerning network traffic, we consider the worst case: Each service is located on a different host, i.e., each message in the workflow generates network traffic. As *START* and *END* represent the client executing the workflow, they are located on the same host. Directed edges between nodes represent the flow of messages between the services. Each node, except for *START*, receives 1–3 input messages (*START* does not receive any input message). Each node, except for *END*, generates 1–3 output messages (*END* does not generate any output message). The concrete number of messages received and generated by a service, as well as the data dependencies between service invocations (i.e., the edges in the workflow) are chosen randomly. There is no edge between the *START* and the *END* node.

In our framework, the result of a service invocation is represented by its output parameters. When delivered to the next services in the workflow, these parameters can be encapsulated in messages customized for each receiver or can be delivered as the same message independently of the destination. In our evaluation, we considered both possibilities: In the setting *SameMsg*, each service invocation generates a single output message which is sent to the 1–3 target nodes, whereas in the setting *DistinctMsg*, each service generates 1–3 distinct output messages.

In the case of centralized orchestration, each message transfer involves the centralized coordinator, which we assume to be located on the same host as the nodes *START* and *END*. In the setting *DistinctMsg*, each edge between two nodes that are different from *START* and *END* corresponds to two messages on the network, because the message has to be sent first to the coordinator. The edges from the *START* node as well as the edges to the *END* node correspond to a single message. In the setting *SameMsg*, the situation is different, because each service has to send only a single message to the coordinator. Moreover, a node sending a message to the *END* node may send the same message also

to other nodes without requiring an extra message to the coordinator (as the *END* node is located on the same host as the coordinator).

In the case of decentralized orchestration using triggers, each edge corresponds to a single message (the edges from the *START* node correspond to client–trigger communication, all other edges represent trigger–trigger communication). We assume that each trigger is located on the same host as the service it invokes. I.e., trigger-service communication does not generate network traffic. However, client-trigger communication (in order to create triggers) causes additional messages, proportional to the number of service invocations in the workflow. For all measurements, we assume trigger creation messages to be 2KB. We do not use the input-data argument of the `CreateTrigger` primitive, but we assume that input data originating from the *START* node is sent by the `SendData` primitive. Hence, 2KB is a reasonable size for trigger creation messages.

We generated 10 000 random workflows, and for each of them, we computed the network traffic for different settings (*SameMsg* versus *DistinctMsg*), different orchestration schemes (centralized versus decentralized), and varying size of input/output messages (message size between 1KB and 150KB). Each measurement represents the average network traffic (arithmetic mean) computed over the 10 000 random workflows.

Figure 2.3 depicts the total network traffic caused by the different orchestration schemes, whereas Figure 2.4 shows the percentage of network traffic caused by decentralized orchestration using triggers relative to the network traffic caused by centralized orchestration. The results confirm that our decentralized orchestration scheme is able to significantly reduce network traffic when compared with centralized orchestration. In the setting *SameMsg* (resp. in the setting *DistinctMsg*), decentralized orchestration causes about 76% (resp. 59%) of the network traffic due to centralized orchestration for input/output messages larger than 20KB. For smaller input/output messages, there is less reduction of network traffic in the decentralized orchestration scheme. Only for very small input/output messages (1KB), decentralized orchestration using triggers may cause extra network traffic of up to 45% (in the setting *SameMsg*), which is due to the overhead of trigger creation messages.

2.7 Related Work

There is a large amount of related work concerning workflow systems and the decentralized execution of workflows. For instance, reference [32] describes a workflow trading system using mobile agents. More recently, the AMOR system [24] uses mobile agents, too. With the aid of mobile agents, it is possible to move the control of the workflow execution to different sites, which can help to reduce the network bandwidth used by communicated (intermediary) results. Moreover, mobile code enables the dynamic deployment of local data processing functions close to the data where it is needed. For example, if the client has to transform intermediary results before passing them to another service, the

transformation functionality may be provided by the mobile agent which will perform the transformation where the data originates.

The drawbacks of using mobile agents are increased security risks and usually a high overhead. Accepting mobile agents in the execution environment opens the door to potentially malicious or erroneous code. Most mobile agent systems are based on Java, even though it has been shown that current standard Java runtime environments are not able to protect the host against various kinds of attacks [9, 7]. The workflow trading system presented in [32] relied on the J-SEAL2 kernel [6] to protect the host from malicious mobile agents. The J-SEAL2 kernel offers operating system functionality, such as strong isolation, safe termination, and mediated communication, on top of standard Java runtime systems by means of extended program verification and transformation. In order to prevent denial-of-service attacks, the environment was enhanced with a portable resource control mechanisms [8]. The disadvantages of this approach are limitations in the programming model, as well as significant overhead due to the extra verification and transformation. In the future, safe language execution environments with strong isolation capabilities, such as Java isolates [26], may allow to build more reliable mobile code platforms. Because of security risks, our triggers currently do not support mobile code.

In active networking [39], mobile code is used within network packets in order to customize the routing. In contrast, our approach allows the customization of the routing of service results at the application level.

The internet indirection infrastructure i3 uses triggers to decouple sender and receiver [36]. In contrast to our approach, i3 triggers work on the level of individual packets and do not support waiting conditions (synchronization) to aggregate multiple inputs from various locations before forwarding the data. i3 supports only a very limited form of service composition, where individual packets can be directed through a sequence of services. While our triggers are rather transient (used only for a single service invocation) and their placement is explicitly controlled by the client, i3 triggers are more persistent (they act as a longer-term contact point for a service) and are mapped to the Chord [37] peer-to-peer infrastructure, which allows only a limited form of optimizing the routing (by selecting a trigger identifier that will map close to a desired location). Summing up, even though there are some ideas in common, i3 has different goals (indirection, supporting mobility, multicast, anycast) and works at a much lower level than our approach. Our focus is on the efficient routing of intermediary results during the execution of composite web services.

In reference [30] the authors point out the inefficiencies of the centralized orchestration of BPEL4WS programs [10] by engines such as BPWS4J [11]. They describe an algorithm to decompose BPEL4WS programs for decentralized orchestration. In contrast to this work, which is restricted to the execution of BPEL4WS programs, our service invocation triggers provide a much more generic and lightweight infrastructure that may serve as the basis for different workflow orchestration models and engines.

The SELF-SERV system [5] focuses on web service composition. It supports peer-to-

peer orchestration of composite web services without relying on a centralized coordinator. While reference [5] describes a rather complex middleware, our triggers are a lightweight solution that can be easily integrated into existing infrastructure.

```
SID0 = createLocalService("result");
SID1 = SID2 = locateService("getGPS");
SID3 = locateService("getRoute");
SID4 = locateService("tiff2jpeg");

TID0 = CreateTrigger:
   destination = location(SID0), // destination = localhost
   SID = SID0,
   input = [("result", required, preserveLast)],
   inputData = [],
   output = [];

TID4 = CreateTrigger:
   destination = location(SID4),
   SID = SID4,
   input = [("image_tiff", required, preserveLast)],
   inputData = [],
   output = [("image_jpeg" -> [(TID0, "result")])];

TID3 = CreateTrigger:
   destination = location(SID3),
   SID = SID3,
   input = [("from_gps", required, preserveLast),
            ("to_gps",   required, preserveLast)],
   inputData = [],
   output = [("route_tiff" -> [(TID4, "image_tiff")])];

TID2 = CreateTrigger:
   destination = location(SID2),
   SID = SID2,
   input = [("adr_text", required, preserveLast)],
   inputData = [("adr_text" <- destination)],
   output = [("adr_gps" -> [(TID3, "to_gps")])];

TID1 = CreateTrigger:
   destination = location(SID1),
   SID = SID1,
   input = [("adr_text", required, preserveLast)],
   inputData = [("adr_text" <- start)],
   output = [("adr_gps" -> [(TID3, "from_gps")])];
```

Figure 2.2: Executing a composite service using triggers.

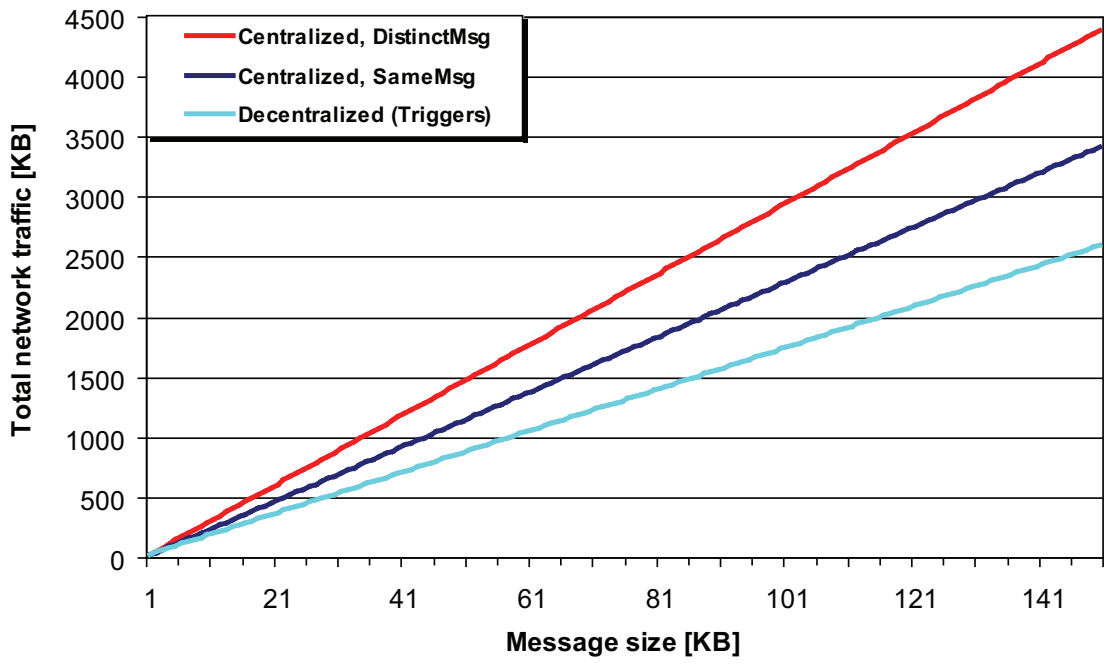

Figure 2.3: Total network traffic using centralized orchestration (settings *SameMsg* and *DistinctMsg*) resp. decentralized orchestration (average computed over 10 000 random, acyclic workflows).

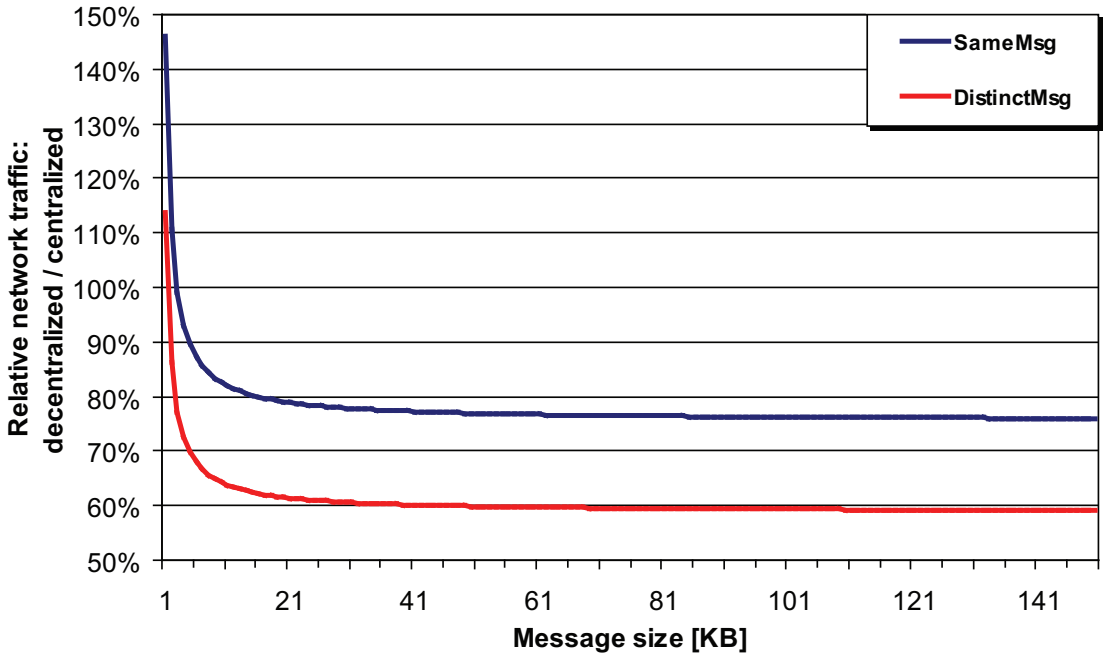

Figure 2.4: Percentage of network traffic with decentralized orchestration using triggers relative to centralized orchestration (average computed over 10 000 random, acyclic workflows).

Chapter 3

Coordination of Services

3.1 Background

Coordination is possibly the defining problem of cooperative working and is essential when the activities that agents perform can interact in any way. The coordination problem is concerned with how to *manage interdependencies between the activities of agents*. Consider the following real-world examples.

- *Jerry and George want to leave a room, and so they independently walk towards the door, which can only fit one person through at a time. Jerry graciously permits George to leave first.* In this example, the activities need to be coordinated because there is a resource (the door) which both people wish to use, but which can only be used by one person at a time.
- *George intends to submit a grant proposal, but in order to do this, he needs Jerry's signature.* In this case, George's activity of sending a grant proposal depends upon Jerry's activity of signing it off – George cannot carry out his activity until Jerry's is completed. In other words, George's activity *depends* upon Jerry's.
- *Jerry obtains a soft copy of a paper from a Web page. He knows that this report will be of interest to George as well. Knowing this, Jerry pro-actively photocopies the report, and gives George a copy.* In this case, the activities do not strictly need to be coordinated – since the report is freely available on a Web page, George could download and print his own copy. But, by pro-actively printing a copy, Jerry saves him some time.

Notice that coordination, defined in this way, encompasses the well-known (and widely studied) concept of *synchronisation* [4]. Synchronisation is generally concerned with the rather restricted case of ensuring that processes do not destructively interact with one another. While solving this problem certainly requires coordination, the concept of

coordination is actually much broader than this. Standard solutions to synchronisation problems involve *hard-wiring* coordination regimes into program code. Thus, for example, a programmer may flag a Java method as `synchronized`, indicating that a certain access regime should be enforced whenever the method is invoked. However, in large-scale, open, dynamic systems, such hard-wired regimes are too limiting. We ideally want computational processes to be able to *reason about* the coordination issues in their system, and resolve these issues *autonomously*.

In order to build agents that can reason about coordination issues dynamically, we must first identify the possible interaction relationships that may exist between the agents' activities. Hence, the goal is to derive and formally define the possible interaction relationships that may exist between activities. Some prior work on this topic exists — von Martial [42] puts forward a high-level typology for coordination relationships. He suggested that, broadly, relationships between activities could be either *positive* or *negative*. Positive relationships "are all those relationships between two plans from which some benefit can be derived, for one or both of the agents plans, by combining them" [41, p. 111]. In other words, positive relationships lead to an increase in the quality of the solution or utility of participants whereas negative relationships lead to a reduction in the quality of the solution or utility of the participants. Such relationships may be *requested* (one agent *explicitly* asks another for help with its activities) or *non requested* (it so happens that by working together mutliple agents can achieve a solution that is better for at least one of them, without making the other any worse off). Von Martial distinguishes three types of non-requested relationships:

The action equality relationship: Jerry and George plan to perform an identical action and, by recognizing this, one of them can perform the action alone, thereby saving the other some effort.

The consequence relationship: The actions in Jerry's plan have the side-effect of achieving one of George's goals, thus relieving George of the need to explicitly achieve it.

The favour relationship: Some part of Jerry's plan has the side effect of contributing to the achievement of one of George's goals, perhaps by making it easier (e.g., by achieving a precondition of one of the actions in it).

Another major body of work on this issue is that on *Partial Global Planning* (PGP) [21]. The basic idea of PGP is that an agent can represent the activities it intends to perform as a plan. It then exchanges this plan of local activity with other agents in order to identify possible interactions (positive or negative). Changes to one or more plans can then be proposed in order to improve performance and the planned local activities are modified in accordance with the coordinated proposal. This work led Durfee to propose the Common Representation for Coordination Hypothesis which stated that "organizations, plans and schedules have a common representation, but differ in their

degree of specificity along different descriptive dimensions." [20]. He termed this common representation a *behavior* and included amongst the descriptive dimensions: what the behaviour was intended to achieve, how it would attempt to achieve it, who was participating in the behaviour, when the behaviour would occur, and why the behaviour has been instituted.

The ideas of PGP were refined in Decker's subsequent work on *Generalised Partial Global Planning* (GPGP) in the TÆMS testbed [18]. GPGP focuses on coordination while agents are scheduling their activities rather than when they are planning to meet goals. Whereas in PGP agents exchange complete schedules at a fixed level of abstraction, in GPGP agents exchange scheduling commitments to particular tasks at any level of abstraction. It utilises domain dependent mechanisms for detecting and predicting coordination relationships and domain independent mechanisms to manage them (by posting constraints to the local scheduler). Five techniques are used for coordinating activities:

- *Updating non-local viewpoints*: Agents have only local views of activities so sharing information can help them achieve broader views. In his TÆMS system, Decker uses three variations of this policy: communicate no local information, communicate all information, or an intermediate level.

- *Communicate results*: Agents may communicate results in three different ways. A minimal approach is where agents only communicate results that are essential to satisfy obligations. Another approach involves sending all results. A third is to send results to those with an interest in them.

- *Handling simple redundancy*: Redundancy occurs when efforts are duplicated. This may be deliberate – an agent may get more than one agent to work on a task because it wants to ensure the task gets done. However, in general, redundancies indicate wasted resources and are therefore to be avoided. The solution adopted in GPGP is as follows. When redundancy is detected, in the form of multiple agents working on identical tasks, one agent is selected at random to carry out the task. The results are then broadcast to other interested agents.

- *Handling hard coordination relationships*: "Hard" coordination relationships are those that threaten to prevent activities being successfully completed. Thus a hard relationship occurs when there is a danger of the agents' actions destructively interfering with one another, or preventing each others actions being carried out. When such relationships are encountered, the activities of agents are rescheduled to resolve the problem.

- *Handling soft coordination relationships*: "Soft" coordination relationships include those that are not "mission critical", but which may improve overall performance. When these are encountered, then rescheduling takes place, but with a high degree of ' negotiability": if rescheduling is not found possible, then the system does not worry about it too much.

Another body of work was performed by Singh, who proposed an event-based linear temporal logic for scheduling service calls [33]. This can be used to provide guards on events, thereby enabling events to be ordered and to be permitted or not based upon the occurence of other events. The approach can be used to enforce coordination relationships such as:

- *enables*: event f cannot occur unless event e occurs beforehand.
- *conditionally feeds*: if events e and f both occur then e occurs before f.
- *guaranteeing enables*: event f can only occur if event e has occured or will occur.
- *initiates*: event f occurs if and only if event e precedes it.
- *jointly require*: if events e and f occur in any order then event g must also occur (in any order).
- *compensates*: if event e occurs and event f does not then event g must be performed.

Singh also stated an important consideration for designing coordination mechanisms: "there is a trade-off between reducing heterogeneity and enabling complex coordination." [33, p. 282] So the more detail of tasks that is given, the better the coordination mechanisms that can be designed but the less widely applicable those mechanisms will be. When designing a general purpose coordination mechanism then, it is best to focus on the most widely shared attributes of tasks. From these a core set of coordination mechanisms can be designed, which can be extended with more domain specific mechanisms."

In related work, WS-Coordination [28] specifies a coordination service consisting of three kinds of sub-service: an Activation Service used by service providers to create the coordination context of their service; a Registration Service used by service requesters to inform the coordination service of their future need for the service; and several Protocol Services that perform the actual coordination. Essentially, it describes what a coordination service should look like and how to interact with it (in particular, describing the messages to be used in such interactions), but nothing is said about how the Protocol Services should perform the actual coordination.

Based on all this body of work, an ontology for coordination was designed, which is presented in the next section. Although ontologies for service based computing have been developed, such as OWL-S [13] and Web Services Modelling Ontology (WSMO) [12], they mainly focus on describing the services and their orchestration/composition.

We argue that our ontology is complementary to existing efforts. Coordination is indeed an important aspect of service based computing, however it addresses the way in which *independent*, and possibly conflicting agents choreograph with others. While in efforts like OWL-S and WSMO the interaction and composition of processes are modelled as a workflow that is determined *a priori* and that is executed by a workflow execution

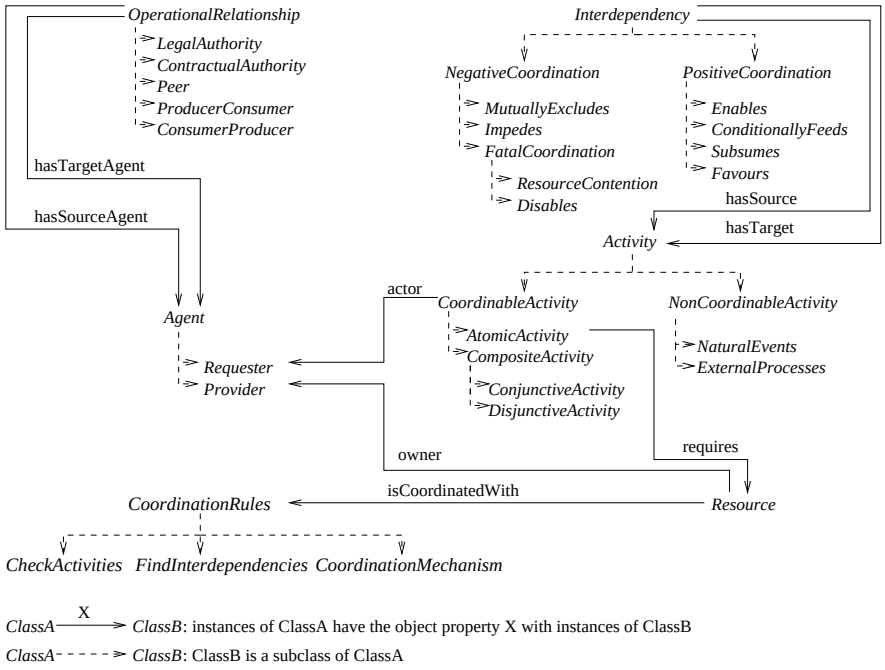

Figure 3.1: An overview of the coordination ontology

component, in agent-based coordination, the choreography is determined by the exchange of messages among the agents that need to interact (*protocol*). However, the first order logic representation of process theory in OWL-S, based on PSL [1], could be integrated in our ontology, in a future implementation [1]. .

3.2 Coordination Ontology

Figure 3.1 provides an overview of the coordination ontology, illustrating the main concepts and relationships. The basic idea is to enable agents to reason about the relationships of their activities to the activities of other agents. So, the fundamental purpose of the ontology is to answer the following questions:

- what is a *coordinable activity*?
- what *coordination relationships* such activities have to one another?

The sub-sections that follow describe the ontology: the key concepts, the slots associated with these concepts, the relationships between these concepts, and axioms. In the

[1]The ontology for coordination was developed in the FP6 EU project Ontogrid (FP6-511513), where a mechanism for coordinating grid services under bounded constraints was developed. The coordination services deployed in Ontogrid relied on a set of rules representing the constraints deriving from the use within Grid environments, such as the use of stateful resources.

interests of comprehensibility, not all of the components of the ontology are presented: the aim is to provide a good overview of the ontology, rather than present all the low-level technical details.

3.2.1 Agents

Our starting concept is *Agent*, which relates to the agents in the system, i.e., the things that do the actions in the system needing to be coordinated. For the purposes of the coordination ontology, agents have just one slot: id, which is a string representation of the unique identifier for the agent (e.g., a URI). Agents can provide or consume a *resource*. To this end, there are two subclasses of agent, *provider* and *requester*. As an agent may be simultaneously both a provider and a requester these subclasses are not disjoint from one another.

3.2.2 Resources

The *Resource* concept describes resources that may be required to expedite an activity. It has the following slots:

- $viable$: a Boolean value, indicating whether the resource is still in a state to be used; a value of `false` here would indicate that the resource could not be used by any activity (even if these activities require it). Another simple way to think about $viable$ is that it indicates whether a resource is "working" or "broken".

- $consumable$: a Boolean value, which indicates whether the use of the resource will reduce subsequent availability of the resource in some way; more precisely, whether the repeated use of the resource in activities would make the resource non-viable.

- $shareable$: a Boolean value, indicating whether a resource may be used by more than one agent at any given time.

- $cloneable$: a Boolean value, indicating whether or not the resource is cloneable (= `true`), or unique and not-cloneable (= `false`). An example of a cloneable resource would be a dataset or a digital document. An example of a unique resource would be a physical artefact produced as the output of a particular experiment, or a human being.

- $owner$: either an $Agent$ (in which case this is the agent that owns the resource), or `null` (in which case the semantics are that the resource may be used by any agent at no cost). If a resource is owned by an agent, and another agent wishes to use this resource, then it may be necessary to enter into negotiation over the exploitation of the resource.

3.2.3 Processes and Activities

The next concept is *Activity*, whose definition was influenced by the OWL-S model of processes [13]. It represents an activity that changes the state of the environment in some way. It may be terminating or non-terminating, and be carried out by a human or other agent, or be a natural (physical) process.

The activity concept has two sub-classes: the most important of which is that of a *CoordinableActivity*. A coordinable activity is a process that can be managed in such a way as to be coordinated with other coordinable activities. For example, executing the process of invoking a web service would be a coordinable activity, in the sense that the invocation of such a service can be managed so as to coordinate with other invocations. For example, suppose there are two agents, both of which want to invoke the same web service, with different parameters. Then, in general, the agents could manage their invocations so as not to interfere with one another.

Not all processes of interest to a system are coordinable – hence the *NonCoordinableActivity* concept. This concept is intended to capture all those processes whose coordination is not possible by the agents within the system to which a particular knowledge base refers. This will include at least the following two types of process:

- *Natural events*: These are physical processes that will take place irrespective of what any agent in the system does. An extreme example would be the decay of an atom, caused by essentially random quantum events. Clearly, such processes cannot be coordinated with other processes: they will take place (or not take place) irrespective of what the agents in the system do.
- *External processes*: These are processes – either physical world processes or natural processes – which are simply outside the control of the system, in that they cannot be managed by the agents in the system. Notice that such processes may be coordinated by entities *outside* the system: the point is, that for the purposes of the system to which the knowledge base refers, they cannot be coordinated.

Another way of thinking about the distinction between a coordinable and a non-coordinable activity is that there is always an agent (i.e., a software agent within the system) associated with a coordinable activity, whereas there is no such agent associated with a non-coordinable activity.

A *CoordinableActivity* will have the following slots:

- *actor*: an $Agent$, i.e., the agent that intends to carry out, or has carried out this activity;
- *earliest start date*: either a date or `null`, with a date indicating the earliest date at which the activity may begin; `null` indicates that this information is not known;

- *latest start date*: either a date or `null`, with a date indicating the latest date at which the activity may begin; `null` indicates that this information is not known;
- *expected duration*: either a natural number, indicating the number of milliseconds the activity is expected to take, or `null` indicates an unknown duration;
- *latest end date*: either a date or `null`, with a date indicating the latest date at which the activity may end; `null` indicates that this information is not known;
- *actual start date*: either a date or `null`, with a date indicating the date at which the activity actually began or is scheduled to begin; `null` indicates that this information is not known;
- *actual end date*: either a date or `null`, with a date indicating the date at which the activity actually ended or is scheduled to end; `null` indicates that this information is not known;
- *shareable result*: a Boolean indicating whether the result of the activity can be shared with other agents;
- *status*: an enumeration type, which takes a value as follows: An activity begins by being *requested* and then becomes *scheduled* if no coordination is required or *proposed* if a change has been proposed. If the activity starts before its earliest start date or after its latest start date or if it ends after its latest end date then it is *outOfBounds*. If the results of the activity are available elsewhere then it is *superfluous*. If the activity is no longer needed for some reason then it is *redundant*. When the activity is performed it is *continuing* though it may become *suspended*. When the activity finishes it must have either *failed* or *succeeded*.

There are two direct sub-classes of coordinable activity: *AtomicActivity* and *CompositeActivity*. An atomic activity is the most basic type of activity and is indivisible into other activities. It has an additional property *requires*, which states the resource that it requires. A composite activity is one which is made up of other coordinable activities. Thus, they can be viewed as being arranged into an and/or tree hierarchy of coordinable activities (atomic or composite), with atomic activities as leaves of the tree. A slot *composedOf* contains the list of sub-activities. There are two sub-classes of composite activity:

- *ConjunctiveActivity*: a composite activity that succeeds if all of its sub-activities succeed
- *DisjunctiveActivity*: a composite activity that succeeds if any one of its sub-activities succeeds

Though these two classes may be used directly to implement coordination mechanisms, it is generally more useful to extend them by creating further subclasses with additional semantics. This is illustrated in practice in Section 3.5.

3.2.4 Interdependencies Between Activities

The *Interdependency* concept is used to describe the various inter-relationships that can exist between activities. The semantics of this concept are based on the work discussed in Section 3.1. Thus, there are two subclasses:

- *NegativeCoordination*: an interaction which, if it occurs, will lead to a reduction in the quality of the solution or the utility of the participants;
- *PositiveCoordination*: an interaction which, if it occurs, will lead to an increase in the utility of the participants or the quality of the solution.

- and the following set of slots:

- $source$ and $target$: both slots are $Activities$, the idea being that these are the two activities which are interdependent.
- $type$: an enumeration, which indicates whether the relation is "soft" or "hard", with the following semantics:
  - a *hard* relation is one which will materially affect the success or otherwise of the activities;
  - a *soft* relation is one which *may* affect the activities, positively or negatively, but will not affect whether they are successful or not.

Subclasses of *NegativeCoordination* include:

- $MutuallyExcludes$: an instance of this relationship will exist between two atomic activities iff:
  1. they both $Require$ some resource r,
  2. the actual or scheduled usage of r by both activities overlaps;
  3. r is non-shareable.

  The idea is thus that these two activities will be mutually exclusive, in the sense that they cannot possibly both succeed as scheduled, as they require access to a resource that cannot be shared. The *type* of this interdependency is therefore *hard*.
- $Impedes$: an instance of this relationship will exist between two $AtomicActivity$s iff:
  1. they both $Require$ some resource r,

2. the actual or scheduled usage of r by both activities overlaps;
3. r is shareable.

The idea is thus that these two activities will impede one another though they will not necessarily prevent each other from succeeding. The *type* of this interdependency is *soft* as it need not necessarily be managed for the system to run effectively.

There is a further sub-class of *NegativeCoordination*: *FatalCoordination* is a hard coordination relationship which, if it occurs, will inevitably lead to the failure of one or more of the component activities. Note that instances of $FatalCoordination$ relationships are always $hard$. Sub-classes of *FatalCoordination* include:

- $Disables$: one activity will disable another if the occurrence of it will definitively prevent the occurrence of the other. This is a *hard* interdependency.
- $ResourceContention$: an instance of this relationship will exist between two atomic activities iff:

  1. they both require some resource r;
  2. resource r is consumable.

  The idea here is thus that one of the activities (the earlier one) could prevent the successful completion of the other activity, by depleting it or rendering it unviable. $ResourceContention$ relationships are not required to be $hard$ although, of course, they could be.

Sub-classes of *PositiveCoordination* are:

- *ConditionallyFeeds*: in such an interdependency, the occurrence of activity A_1 will subsequently make possible the occurrence of activity A_2, but it is nevertheless possible that A_2 could occur (i.e., the occurrence of A_1 is a sufficient but not necessary event for the occurrence of A_2). This is a *hard* interdependency.
- *Enables*: the occurrence of activity A_1 is both necessary and sufficient for the occurrence of A_2. This is a *hard* interdependency.
- *Subsumes*: activity A_1 subsumes activity A_2 if A_1 contains all the activities of A_2. This is a *soft* interdependency.
- *Favors*: an activity A_1 favors another activity A_2 if its prior occurrence will subsequently improve the overall quality of A_2. We include this as a "catch all". This is a *hard* interdependency.

3.2.5 Operational Relationships

In order to *resolve* a coordination relationship between two activities, it may be necessary to appeal to the *operational relationships* that exists between the agents that will carry them out. Intuitively, operational relationships exist between agents, and by understanding these relationships, it can help to resolve interdependencies. The main concept then is *OperationalRelationship*. This concept has two slots, both of which are $Agents$: $source$ and $target$. Sub-classes of *OperationalRelationship* include:

- *LegalAuthority*: this sub-class indicates that $source$ has legal authority over $target$ (of course, this begs the question of what "legal authority" means in the context of semantic web services and processes, but this is outside the scope of our current work, and is left as a placeholder for the future);
- *ContractualAuthority*: this indicates that $source$ has contractual authority over $target$ (i.e., that both agents "belong" to the same organisation, and that in the context of this organisation, $source$ should take precedence over $target$);
- *ProducerConsumer*: this indicates that $source$ is the $owner$ of a $Resource$ that is to be used by $target$;
- *ConsumerProducer*: the inverse of $ProducerConsumer$;
- *Peer*: two agents that work as peers, i.e., that neither has any authority over the other.

3.3 Coordination Rules

The ontology provides a means of describing activities and the interdependencies that may exist between them. This knowledge can then be used to coordinate the various activities with one another. For this purpose, a number of rules were developed. For the sake of clarity, they are split into three groups:

- Rules to check activities
- Rules to detect interdependencies between activities
- Rules to manage interdependencies between activities

The sets of rules can be seen as building upon one another. The first set ensure that the descriptions of activities are complete and consistent. The second set then use these consistent description to identify any interdependencies that exist between activities. Finally, the third set take the interdependencies identified and manage them accordingly.

3.3.1 Rules to Check Activities

These rules are used to check activities and detect any inconsistencies or omissions. Essentially they are used to capture some of the basic axiomatic properties of the ontology. This entails that whenever a new type of activity is added to the ontology it may be necessary to add some new rules to this set.

Rules to Check All Coordinable Activities

1. If an activity's latest start date is after or the same as its latest end date, then set the latest start date to be the latest end date - the expected duration
2. If an activity's actual end date is not its actual start date + expected duration, then change the actual end date accordingly
3. If an activity started before its earliest start date, then its status should be set to 'outOfBounds'
4. If an activity started after its latest start date, then its status should be set to 'outOfBounds'
5. If an activity started after its latest end date, then its status should be set to 'outOfBounds'
6. If an activity ended after its latest end date, then its status should be set to 'outOfBounds'
7. If an activity's earliest start date is after or the same as its latest end date, then its status should be set to 'outOfBounds'
8. If an activity's earliest start date is after its latest start date, then its status should be set to 'outOfBounds'

Rules to Check Composite Activities

1. If a composite activity does not have an enables or conditionally feeds interdependency with another activity then its actual start date should be that of the component activity with the earliest actual start date. (The expected duration is also modified accordingly.)
2. The actual end date of a composite activity should be that of the component activity with the latest actual end date. (The expected duration is also modified accordingly.) This rule assumes a pessimistic view for disjunctive activities, i.e. they always take the longest time possible.

Rules to Check Component Activities

1. If a component activity is part of a composite activity which has an enables or conditionally feeds interdependency with another activity then actual start date of the component activity should not be earlier than the actual start date of its composite activity.

2. If a composite activity has status 'failed', 'succeeded' or 'redundant' then all subactivities should have the status 'redundant'.

3. If the earliest start date of a component activity is before that of the composite activity then it is set to the latter.

4. If the latest end date of a component activity is after that of the composite activity then it is set to the latter.

5. If the latest start date of a component activity is after the latest end date of the composite activity then it is set to the latter.

6. If the latest start date of a component activity is before the earliest start date of the composite activity then its status is set to 'failed'.

7. If the expected duration of a component activity is greater than the difference between the earliest start date and latest end date of the composite activity then its status is set to 'failed'.

Rules to Check Conjunctive Activities

1. If all component activities of a conjunctive activity have status 'succeeded' then set the status of the conjunctive activity to 'succeeded'.

2. If any one of the component activities of a conjunctive activity has status 'failed' then set the status of the conjunctive activity to 'failed'.

Rules to Check Disjunctive Activities

1. If any one of the component activities of a disjunctive activity has status 'succeeded' then set the status of the disjunctive activity to 'succeeded'.

2. If all component activities of a disjunctive activity have status 'failed' then set the status of the disjunctive activity to 'failed'.

if
(AtomicActivity A_1 (requires $resource_r$, actualStartDate SD_1, actualEndDate ED_1))
(AtomicActivity A_2 (requires $resource_r$, actualStartDate SD_2, actualEndDate ED_2))
($resource_r$ (shareable false))
($SD_2 > SD_1$)
($ED_2 > ED_1$)
($ED_1 > SD_2$)
then
(MutuallyExcludes (hasSource A_1, hasTarget A_2, hasDuration (ED_2-SD_1)))

Figure 3.2: Example rule to find an interdependency

3.3.2 Rules to Detect Interdependencies Between Activities

These rules examine activities and infer new instances of interdependencies from them. Similar to the rules above, this entails that whenever a new type of activity is added to the ontology new rules may need to be added to this set. If a new type of interdependency is added to the ontology then new detection rules will certainly be required.

The rules themselves are divided into two categories: those that detect positive interdependencies and those that detect negative interdependencies. Negative interdependencies tend to be more general and as such the rules to find them are readily applicable to a wide range of domains. Positive interdependencies on the other hand tend to be more domain-specific, relying on particular properties of activities, and hence specific rules have to be written to find such interdependencies. Section 3.5 demonstrates how rules can be created tailored to a particular domain. The following rules are currently implemented to detect instances of the negative interdependencies *impedes* and *mutually excludes*:

1. If the end of a time slot for an activity overlaps the beginning of the time slot for another activity, and both activities require access to the same non-shareable resource, then assert an interdependency stating that the two activities are mutually exclusive. This rule is illustrated in Figure 3.2.

2. If the time slot for an activity is included in the time slot for another activity, and both activities require access to the same non-shareable resource, then assert that the two activities are mutually exclusive.

3. If the end of a time slot for an activity overlaps the beginning of the time slot for another activity, and both activities require access to the same shareable resource, then assert an interdependency stating that the two activities impede one another.

4. If the time slot for an activity is included in the time slot for another activity, and both activities require access to the same shareable resource, then assert that the two activities impede one another.

3.3.3 Rules to Manage Interdependencies Between Activities

These rules describe the coordination regime itself. Thus, different sets of rules can be used to provide different regimes, depending upon requirements.

Each of the rules acts on an interdependency by modifying a coordinable activity accordingly. This modification consists of changing either the actual start date, the actual end date or the status of the activity. In the case of the start or end date being modified, the knowledge base is first queried to find an suitable new slot. Once an interdependency has been managed, it is removed along with all other unmanaged interdependencies involving the modified activity. This ensures that the knowledge base is left in a consistent state.

The coordination rules themselves are based upon operational relationships first and foremost. So, for example, if two agents related by a legal authority relationship request to carry out conflicting activities then the activity of the agent with the lower precedence is modified. A similar rule exists for when there is a contractual authority relationship. If the two agents are peers or the same agent requests two conflicting activities then the shortest activity is moved. This is one example of a coordination regime though it could be readily substituted for another.

Additionally, it was determined that it would be appropriate to distinguish between hard and soft interdependencies so that hard interdependencies, which determine the successful execution of the system, are always handled before soft interdependencies, which only affect efficiency. For this reason, each rule is effectively replicated with the only difference occuring in the type of interdependency encountered. It is then possible to specify that the rules dealing with hard interdependencies have priority over those dealing with soft interdependencies.

The following rules have been implemented to provide a working example of a coordination regime:

1. If two activities are mutually exclusive and they were requested by different agents, one of which has a legal authority over the other, then move the activity of the agent with lower precedence.

2. If two activities are mutually exclusive and they were requested by different agents, one of which has a contractual authority over the other, then move the activity of the agent with lower precedence.

3. If two activities are mutually exclusive and they were both requested by the same agent or they were requested by different agents, neither of which has authority over the other (i.e. they are peers or no relationship between the two has been explicitly stated), then move the activity with the smallest expected duration. If they are both of equal expected duration then move the activity that was requested last.

4. If two activities impede one another and they were requested by different agents,

one of which has a legal authority over the other, then move the activity of the agent with lower precedence.[2]

5. If two activities impede one another and they were requested by different agents, one of which has a contractual authority over the other, then move the activity of the agent with lower precedence.

6. If two activities impede one another and they were both requested by the same agent or they were requested by different agents, neither of which has authority over the other (i.e. they are peers or no relationship between the two has been explicitly stated), then move the activity with the smallest expected duration. If they are both of equal expected duration then move the activity that was requested last.

7. If an activity enables another activity then modify the latter activity so that it occurs after the activity that enables it.

8. If an activity conditionally feeds another activity then modify the latter activity so that it occurs after the activity that conditionally feeds it.

9. If an activity subsumes another activity then set the status of the subsumed activity to 'superfluous'.

10. If an activity that subsumes another activity succeeds then set the status of the subsumed activity to 'succeeded'. This only occurs if the subsumed activity still has the status 'superfluous' as it may have become redundant or be part of a composite activity that is redundant, failed or succeeded.

11. If an activity that subsumes another activity fails then set the status of the subsumed activity to 'requested', i.e. attempt to reschedule the subsumed activity. Again, this only occurs if the subsumed activity still has the status 'superfluous'.

12. If an activity that subsumes another activity becomes redundant then set the status of the subsumed activity to 'requested', i.e. attempt to reschedule the subsumed activity. Once more, this only occurs if the subsumed activity still has the status 'superfluous'.

3.4 Implementation

The coordination ontology was implemented in the Web Ontology Language (OWL) [14] using Protégé 3.0 [35] and all of the rules were implemented in Jess [23]. The JessTab [22] plug-in for Protégé was used to enable the ontology to be loaded into a Jess rule engine

[2]Note that it is possible to simply allow both activities to proceed as scheduled, but with their durations increased by a particular factor. This would be an example of an alternative coordination regime.

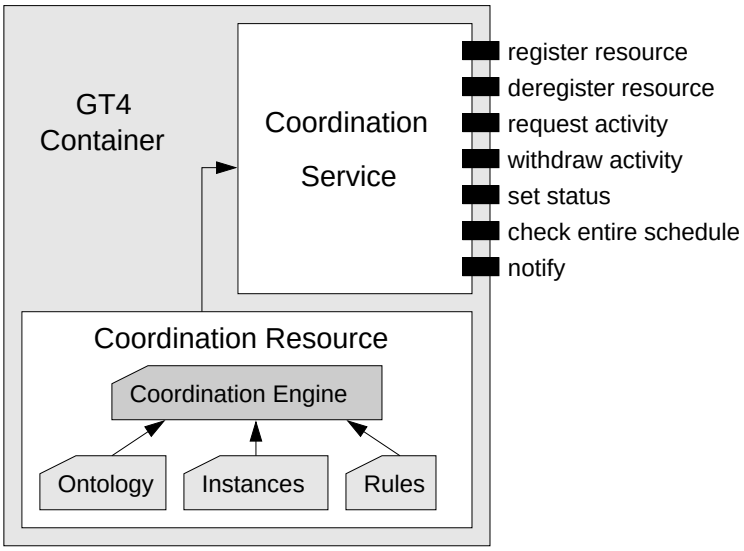

Figure 3.3: Architecture of the coordination service

as a Protégé knowledge base. This effectively encapsulates the coordination engine (ontology, instances, rules and rule engine) into a Java object.

A Web Services that acts as a wrapper for the coordination engine and provides methods the following methods:

- *register resource*: used to register a new resource
- *deregister resource*: used to deregister a resource
- *request activity*: used to request a new activity
- *withdraw activity*: used to withdraw an activity request should it no longer be required
- *set status*: used by the requester of an activity to set the activity's status
- *check entire schedule*: used to retrieve the entire list of activities for a particular resource

Additionally, we made use of WS-Notification [25] in order to implement a publish-subscribe notification mechanism allowing agents to subscribe to receive messages detailing activity changes or the deregistration of resources. This asynchronous form of communication is essential for updating resource providers and requesters of changes which may impact upon them. The service is illustrated in Figure 3.3.

A graphical client application was also developed to allow for intuitive user interaction with the service when performing testing and evaluation. The client provides access to

all of the service's API methods and automatically subscribes to receive all notification messages sent by the service. Using these messages, the client is able to build up a representation of the internal state of the service. The user can then access this information in the form of a Gantt chart representing either the entire list of resources and activities known to the service, or the list of activities for a specific resource. With this information on screen, the user is able to select an activity and view the interdependencies associated with it.

3.5 Use Case

This section presents a sample scenario, taken from the domin of car insurance fraud, to which a centralised coordination mechanism could be applied to succesfully coordinate a number of activities. This scenario was devised for the evaluation of the coordination services deployed in the EU project Ontogrid project (FP6-511513)

The scenario involves a number of insurance companies who wish to collaborate to discover whether the claims they receive are fraudulent. As such, a virtual organisation (VO) is established and member insurance companies make their databases available to other members (though with many limitations). When one of the members then wishes to assess a claim it performs checks against a number of known fraud models. For example, the Berliner fraud model involves stealing a car and then crashing it into an insured car that is already damaged and claiming the damage from the insurance company of the stolen car. Other fraud models include the stolen cars model, the Saarland model and the Autobumser model. To detect whether one of these models is present, the insurance company will send a number of queries to the other insurance companies in the VO and then aggregate and analyse the results. Generally, the models should be checked in a specified order, and if one of the models is detected then the rest need not be checked.

3.5.1 The Ontology

To implement this scenario, the coordination ontology was extended as illustrated in Figure 3.4. Working from the bottom up, two new types of resource are included in the ontology:

- *CPU*: This represents a CPU that will be used to perform some processing task. The property $shareable$ is set to true as multiple processing tasks may be performed simultaneously.
- *InsuranceCompanyBO*: This represents a database (back-office) of an insurance company. The property $shareable$ is set to false as the back-office operations are intensive and the insurance companies wish to limit the number of operations that can be performed.

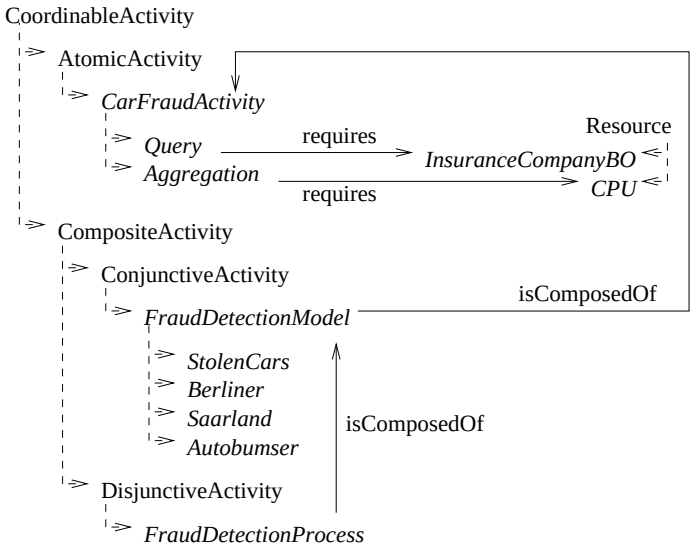

Figure 3.4: The extended ontology for the car fraud use case

There are then two new sub-classes of AtomicActivity which use these resources (though an intermediate class *CarFraudActivity* is introduced for clarity):

- *Query*: These represent the queries that are performed on the insurance company databases, and as such the $requires$ property must be an instance of InsuranceCompanyBO. They also have an additional slot $content$ which details the content of the query.

- *QueryAggregation*: These represent the aggregations of queries that are performed by insurance companies. As such the $requires$ property must be an instance of CPU.

These activities are then used to compose *FraudDetectionModel* activities, which is a type of composite activity representing a fraud detection model. As such it has subclasses for each of the detection models (StolenCars, Berliner, Saarland and Autobumser). FraudDetectionModel is itself a sub-class of ConjunctiveActivity, as it is necessary for all of the queries and query aggregations within a particular model to complete successfully in order for that activity to complete successfully.

Finally, FraudDetectionModel activities are used to compose *FraudDetectionProcess* activities, which is also a type of composite activity and represents the entire process undertaken by an insurance company checking for fraud. It is a sub-class of 'DisjunctiveActivity', since if any of the component fraud detection model activities succeeds then the whole fraud detection process succeeds.

3.5.2 The Rules

With the new activities defined it was necessary to consider whether any new rules are also needed. Such rules may be required in any of the three categories and so the following process was observed.

Firstly, it may be necessary to define new rules to check the consistency and completeness of newly defined activities. Often, however, this will not be the case, since new activities will extend existing activities and so the existing rules will also apply to these new activities.

Next, it is necessary to examine any interdependencies that may involve the new activities to determine whether any new rules are required for detecting them. As stated in Section 3.3.2, the rules to detect negative interdependencies are more generally applicable than those used to detect positive interdependencies, so it is unlikely that new rules will be required here, unless new types of negative interdependency have also been defined in the ontology. It is more likely there will be positive interdependencies which are already classified in the ontology but which rely on the particular properties of domain specific activities. In these cases specialised rules must be written to detect such interdependencies. Of course, new types of positive interdependency may be defined as well, in which case rules will be needed to detect these too.

Finally, it is necessary to determine whether any new rules are required for managing interdependencies. Generally though, this will not be the case unless a new coordination regime is required.

Following this process it was found that no new rules were necessary for checking activities or for managing interdependencies. Additionally, the following interdependencies would be detected by the existing detection rules:

- If two Query activities require the same InsuranceCompanyBO at the same time then they *mutually exclude* one another, since the InsuranceCompanyBO is non-shareable.
- If two QueryAggregation activities require the same CPU at the same time then they *impede* one another, since the CPU is shareable.

However, several new rules were required for detecting interdependencies:

- If a Query activity and a QueryAggregation activity are part of the same FraudDetectionModel then the Query *enables* the QueryAggregation.
- If two Query activities require the same InsuranceCompanyBO and have the same content then the activity with the earliest actual end date *subsumes* the other.
- If a StolenCars activity is part of the same FraudDetectionProcess as a Berliner or Saarland or Autobumser activity then the StolenCars activity *enables* the other.

- If a Berliner activity is part of the same FraudDetectionProcess as a Saarland or Autobumser activity then the Berliner activity *enables* the other.
- If a Saarland activity is part of the same FraudDetectionProcess as an Autobumser activity then the StolenCars activity *enables* the Autobumser activity.

3.6 Evaluation

In initial testing of the system, the coordination service was found to detect and manage all of the expected interdependencies between activities. Given this, a test harness was developed based upon the client described in Section 3.4. This harness allows the user to set a number of variables from which it generates a series of activities to submit to the coordination service. As the test harness subscribes to all of the notifications sent by the coordination service, it has a complete view of the status of the coordination engine at all times. From this it is able to determine if a newly submitted activity should have any interdependencies with an existing activity. It then checks for each new activity whether any interdependencies are detected and whether they are managed successfully.

The following factors were varied in the evaluation:

- The number of resource providers
- The number of resource requesters
- The number of resources
- The number of activities to submit
- The number of interdependencies that should exist between submitted activities

- and the following results were measured:

- The response time of the service
- The number of interdependencies detected/resolved
- The level of communication between the service and its users, i.e. the number of notifications sent in response to a service call. Typically, such notifications will be sent when activities are moved as a result of an interdependency being detected and resolved.
- The number of activities that cannot be scheduled within the bounds of their earliest and latest start/end dates

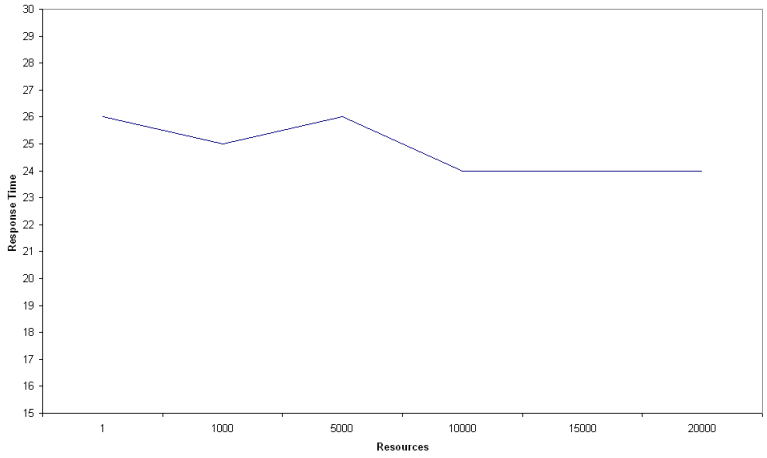

Figure 3.5: Number of Resources vs. Response time

3.6.1 Number of Resources vs. Response Time

The first experiment performed was intended to identify how the system performed with a varying numbers of resources. A series of runs were performed during which a constant number of activities were submitted to the service, whilst the number of providers, requesters and interdependencies were also kept constant.

The only variable was the number of resources being managed by the coordination service. The time taken for the coordination service to respond to requests for new activities was recorded. The results are detailed in Figure 3.5.

As can be seen from these results, the number of resources being managed by the coordination service had no discernible effect on the time taken to respond to service calls to add new activities. Only a difference of 2ms (around 7.5%) existed between the response times of the service with 1 resource and that with 20,000 resources. Furthermore, the service managing 20,000 resources responded more rapidly than that managing a single resource.

The explanation for this is that for the service managing a single resource, all activities were specified as using that one resource, whereas for the service managing 20,000 resources the activities were distributed over all of these resources. Hence when a new activity is added to the service managing one resource, it takes longer to check that activity against the list of other activities already using that resource.

3.6.2 Number of Activities vs. Response Time

The next experiment performed was intended to identify how the system performed with a varying numbers of activities. A series of runs were performed during which a constant

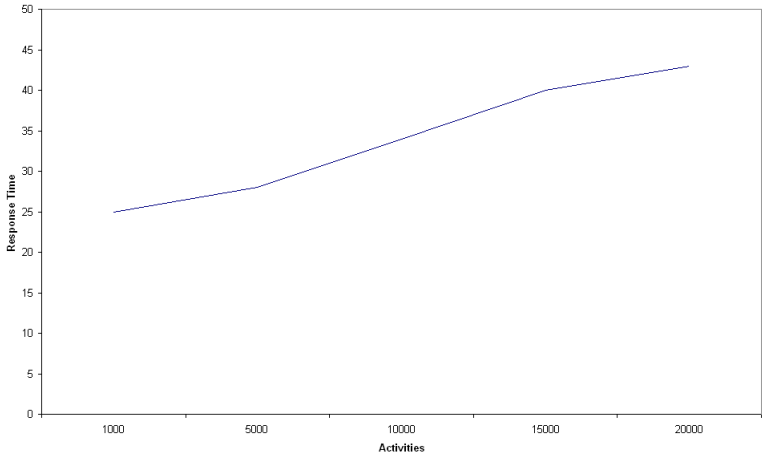

Figure 3.6: Number of Activities vs. Response time

number of resources were managed by the service, whilst the number of providers, requesters and interdependencies were also kept constant. For this experiment, the number of interdependencies was set to zero.

The only variable was the number of activities to add to the coordination service. The time taken for the coordination service to respond to requests for new activities was recorded. The results are detailed in Figure 3.6.

As can be seen from these results, the greater the number of activities being managed by the coordination service, the longer the service took to respond to service calls to add new activities. This relationship was found to be linear, i.e. as the number of activities added was increased, the response time increased proportionately.

This result was as expected and has a similar explanation as the result found in the previous experiment, i.e. with a larger number of activities there are more activities per resource and hence when a new activity is added to the service managing a resource, it takes longer to check that activity against the list of other activities already using that resource.

3.6.3 Number of Interdependencies vs. Response Time

Another experiment was carried out to identify how the system performed with a varying numbers of interdependencies. A series of runs were performed during which a constant number of resources were managed by the service and a constant number of activities were requested using these resources. Additionally the number of providers and requesters were also kept constant. The only variable was the number of interdependencies that should be detected.

The time taken for the coordination service to respond to requests for new activities

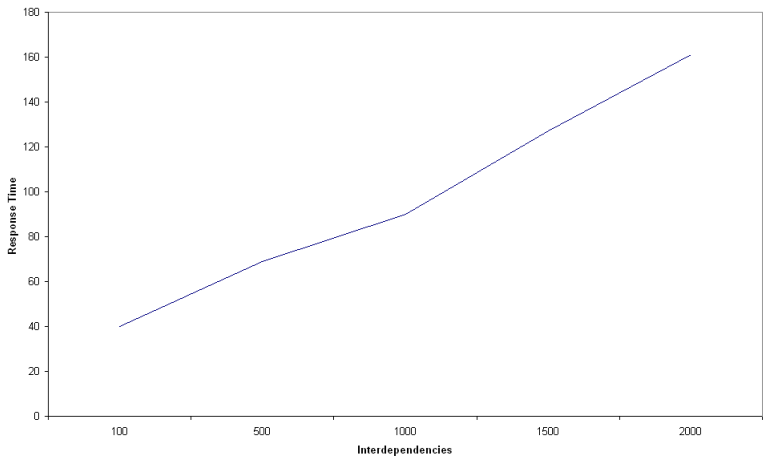

Figure 3.7: Number of Interdependencies vs. Response time

was recorded. This was further divided into the time taken for the service to respond to activity requests when an interdependency was detected and the time taken to respond when no interdependencies were detected. Furthermore, the number of notifications sent by the service and the number of interdependencies detected and resolved were also measured. Finally, the number of activities which were moved as a result of an interdependency such that the start and end dates were now outside the prescribed limits (i.e. that were out of bounds) was also measured. The results are detailed below:

As can be seen from Figure 3.7, the greater the number of interdependencies between activities, the longer it takes for the service to respond to new activity requests. What is more, this relationship appears to be linear. This is to be expected as a call to add a new activity with an interdependency will take longer than one without, since it takes time to manage the interdependency and move any activities appropriately.

However, as illustrated in Figures 3.8 and 3.9, the time taken to respond to activity requests which do not involve an interdependency increases at a much slower rate than the time taken to respond to activity requests with an interdependency. The explanation for this is that as the number of interdependencies increases, and hence the proportion of interdependencies to activities increases, the greater the number of activities that need to be moved around and accommodated when managing the interdependencies. Similarly, this results in a greater proportion of activities which are moved outside of the start and end date limits, as illustrated in in Figure 3.10.

The most significant result of this experiment was that in each run the expected number of interdependencies was detected and resolved. Hence the coordination service consistently identified and resolved all of the different types of interdependency with a 100% success rate.

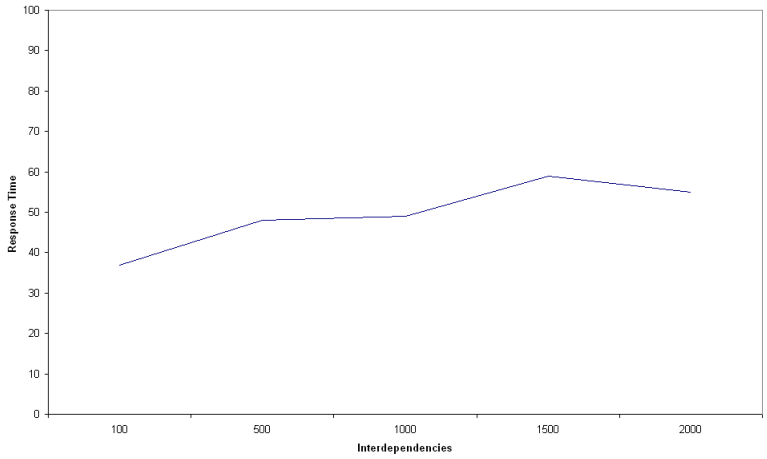

Figure 3.8: Response time for activities without interdependencies

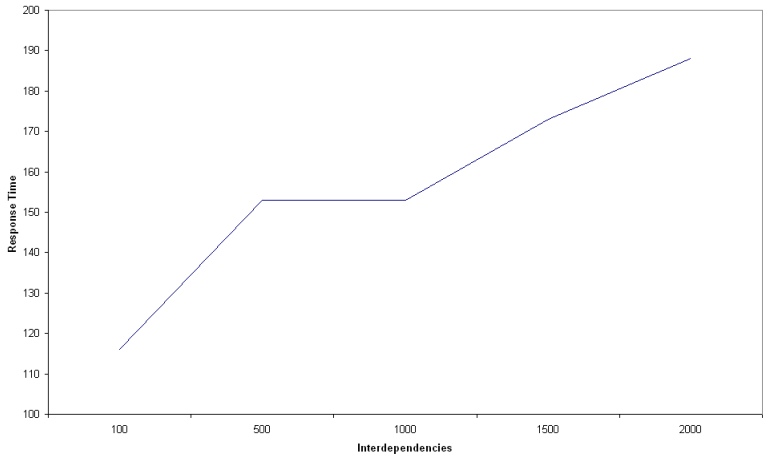

Figure 3.9: Response time for activities with interdependencies

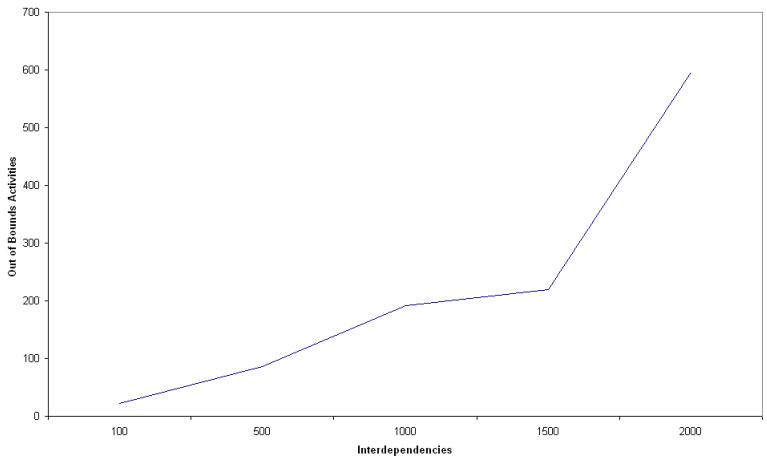

Figure 3.10: Number of Interdependencies vs. Number of 'outOfBounds' activities

3.6.4 Discussion

The experiments demonstrate that the coordination service successfully detects and resolves all of the interdependencies. Furthermore, it is largely unaffected by the number of resources that it has to manage, whilst the response times increase linearly with the number of activities and interdependencies. Also, the proportion of activities that are moved outside of their limits is fairly low until the number of interdependencies approaches the number of activities. Of course, this assumes that activities are randomly added to each resource, as they were in these experiments, and that the flexibility of activities is about 10 times the duration of the average activity (i.e. for an activity of duration 10 time units the flexibility will be 100 time units, or 45 each side of the start and end date). With greater flexibility this proportion will decrease further to a minimum of 0 when no bounds are set for activities.

Several other improvements could be made to increase the efficiency of the coordination mechanism. For example, when an interdependency is detected and an activity is to be moved, the function to find a new slot for that activity currently performs a linear sort and search of all activities using the same resource (within a specified time range). This could be improved by implementing a binary search or interpolation search so as to improve the response time of the system. A number of other rules and functions could also be re-factored for greater efficiency should an industrial strength implementation of the system be required.

Chapter 4

Conclusion

In this deliverable we looked at the integration of independent services, and in particular to the problem of decentralised orchestration of services and we presented some initial work on the provision of run-time coordination mechanisms to manage possible conflicts, or optimize the usage of independent services making use of resources that are scarce or bounded, typical of certain environments such as grids.

In Chapter 2 we described a novel mechanism for orchestration based on service invocation triggers. If a composite web service is executed in a centralized way, intermediary results are forwarded through the site that coordinates the execution. E.g., if a client executes a composite web service on a mobile device with limited network connectivity, the transmission of the intermediary results may significantly slow down the overall execution of the composite web service, it may be expensive (costs for the caused network traffic), or it may be simply impossible if the intermediary results are too large.

In order to overcome these problems, we developed a novel infrastructure with service invocation triggers that are able to route intermediary results from their origin directly to the sites where they are consumed. Triggers act as proxies for individual web service invocations. They aggregate the input data, trigger the service execution when all required input parameters are available (synchronization), and route the service results, supporting multicast. Based on triggers, composite web services can be executed in a completely decentralized way. Evaluation results confirm that our decentralized orchestration scheme allows to significantly reduce network traffic in comparison with centralized orchestration.

The aim of the work presented in Chapter 3 is to provide coordination at run-time rather than being hard-wired at design-time. The solution presented and its implementation demonstrates that the approach developed is a viable means of detecting and resolving interdependencies. In the simplest case the approach can be used in place of a queue-based or prioritized scheduler. When the system becomes more complex, however, and there is a need to dynamically recognize and resolve interdependencies between activities, then traditional queue-based approaches will fail, whereas the approach developed herein will

still manage the resources effectively.

The main point of future is to apply the approach to a decentralized environment. To this end, it would be beneficial to implement the rules in a language such as the Semantic Web Rule Language (SWRL) [2], which would enable them to be encapsulated in the ontology thereby enabling the coordination mechanism to be more portable and exchangeable. However, SWRL currently has a number of limitations which prevent many of the rules being directly translated. For example, it only supports the conjunction of atoms, there is no support for negation and there are no explicit quantifiers but instead implicit universal quantification for all variables. Rule engines for SWRL also suffer from limitations. For example, Bossam citebossam has no built-in support for math functions and string handling, has facility to retract facts (when they are no longer true) and has no support for the boolean datatype.

A related point of future work is the implementation of a number of alternative coordination regimes. These could be collected as librariesso that coordination regimes could be substituted for one another dependent on the environment and requirements. A useful experiment would then be to determine how easily these different regimes could be swapped.

Another point of future work consists of examining the definition of *AtomicActivity* alongside related representations such as those used by BPEL4WS, to see whether it can be made more precise. The key point here though will be that made by Singh, i.e. the more detailed the description of tasks becomes, the better the coordination mechanisms that can be designed but the less widely applicable those mechanisms will be. The mechanism so far developed has been demonstrated to be widely and readily applicable and it would be highly desirable to maintain this level of applicability.

Bibliography

[1] PSL, process specification language. http://www.mel.nist.gov/psl/.

[2] Swrl. http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/.

[3] Apache Software Foundation. Axis, http://ws.apache.org/axis/.

[4] M. Ben-Ari. *Principles of concurrent and distributed programming*. Prentice-Hall, Inc., 1990.

[5] B. Benatallah, Q. Z. Sheng, and M. Dumas. The self-serv environment for web services composition. *IEEE Internet Computing*, 7(1):40–48, 2003.

[6] W. Binder. Design and implementation of the J-SEAL2 mobile agent kernel. In *The 2001 Symposium on Applications and the Internet (SAINT-2001)*, pages 35–42, San Diego, CA, USA, Jan. 2001.

[7] W. Binder. Secure and reliable Java-based middleware – Challenges and solutions. In *First International Conference on Availability, Reliability and Security (ARES-2006)*, pages 662–669, Vienna, Austria, Apr. 2006. IEEE Computer Society.

[8] W. Binder and J. Hulaas. A portable CPU-management framework for Java. *IEEE Internet Computing*, 8(5):74–83, Sep./Oct. 2004.

[9] W. Binder and V. Roth. Secure mobile agent systems using Java: Where are we heading? In *Seventeenth ACM Symposium on Applied Computing (SAC-2002)*, Madrid, Spain, Mar. 2002.

[10] BPEL4WS. Business process execution language for web services version 1.1, http://www.ibm.com/developerworks/library/ws-bpel/.

[11] BPWS4J. A platform for creating and executing BPEL4WS processes, http://www.alphaworks.ibm.com/tech/bpws4j/.

[12] T. S. Cluster. WSMO: Web Services Modelling Ontology, 2004. See http://www.wsmo.org/.

[13] T. O. Coalition. OWL-S: OWL-based Web Service Ontology, 2004. See http://www.daml.org/services/owl-s/.

[14] W. W. W. Consortium. Web site for the specification of OWL, 2004. http://www.w3.org/2004/OWL/ (accessed 31 March 2005).

[15] I. Constantinescu, W. Binder, and B. Faltings. Flexible and efficient matchmaking and ranking in service directories. In *2005 IEEE International Conference on Web Services (ICWS-2005)*, pages 5–12, Florida, USA, July 2005.

[16] I. Constantinescu, B. Faltings, and W. Binder. Large scale testbed for type compatible service composition. In *ICAPS 04 workshop on planning and scheduling for web and grid services*, 2004.

[17] I. Constantinescu, B. Faltings, and W. Binder. Large scale, type-compatible service composition. In *IEEE International Conference on Web Services (ICWS-2004)*, pages 506–513, San Diego, CA, USA, July 2004.

[18] K. Decker and V. Lesser. Designing a family of coordination algorithms. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 73–80, San Francisco, CA, June 1995.

[19] K. Decker and V. R. Lesser. Designing a family of coordination algorithms. In *Proc. of 1st Int. Conf. on MultiAgent Systems (ICMAS-95)*, San Francisco (CA, USA), 1995.

[20] E. Durfee. Organizations, plans, and schedules: An interdisciplinary perspective on coordinating ai systems. In *Journal of Intelligent Systems, Special Issue on the Social Context of Intelligent Systems, 3(2-4)*, 1993.

[21] E. H. Durfee. *Coordination of Distributed Problem Solvers*. Kluwer Academic Publishers: Boston, MA, 1988.

[22] H. Eriksson. Web site for the plug-in JessTab, 2007. http://www.ida.liu.se/~her/JessTab/ (accessed February 2, 2007).

[23] E. Friedman-Hill. *Jess in Action: Java Rule-Based Systems*. Manning Publications Co., 2003.

[24] K. Haller, H. Schuldt, and H.-J. Schek. Transactional Peer-to-Peer Information Processing: The AMOR Approach. In *Proceedings of the 4th International Conference on Mobile Data Management (MDM-2003)*, volume 2547 of *Lecture Notes in Computer Science*, pages 356–361, Brisbane, Australia, Jan. 2003. Springer.

[25] IBM. Web site for WS-Notification, 2007. http://www-128.ibm.com/developerworks/library/specification/ws-notification/ (accessed February 3, 2007).

[26] Java Community Process. JSR 121 – Application Isolation API Specification. Web pages at http://jcp.org/jsr/detail/121.jsp.

[27] T. W. Malone and K. Crowston. The interdisciplinary study of coordination. *ACM Computing surveys*, 26(1):87–119, 1994.

[28] Microsoft, IBM, Hitachi, IONA, Arjuna Technologies, and BEA Systems. Web page for the specification of WS-Coordination, version 1.0 (updated Aug. 2005), 2005. `http://www-128.ibm.com/developerworks/library/specification/ws-tx/` (accessed Oct. 3, 2005).

[29] T. Moyaux, B. Lithgow-Smith, S. Paurobally, V. Tamma, and M. Wooldridge. Towards service-oriented ontology-based coordination. In *Proc. of 4th Int. Conf. on Web Services (ICWS 2006)*, Chicago, IL (USA), 2006.

[30] M. G. Nanda, S. Chandra, and V. Sarkar. Decentralizing execution of composite web services. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 170–187, New York, NY, USA, 2004. ACM Press.

[31] M. P. Papazoglou and D. Georgakopoulos. Introduction: Service-oriented computing. *Communications of the ACM*, 46(10):24–28, Oct. 2003.

[32] M. Schönhoff and H. Stormer. Trading workflows electronically: the ANAISOFT architecture. In *Proceedings of Datenbanksysteme in Büro, Technik und Wissenschaft (BTW'2001)*, pages 67–74, Oldenburg, Germany, Mar. 2001.

[33] M. Singh and M. N. Huhns. *Service-Oriented Computing - Semantics, Processes, Agents*. John Wiley and sons, Ltd., 2005.

[34] M. P. Singh. A customizable coordination service for autonomous agents. In M. P. Singh, A. Rao, and M. J. Wooldridge, editors, *Intelligent Agents IV (LNAI Volume 1365)*, pages 93–106. Springer-Verlag: Berlin, Germany, 1998.

[35] Stanford Medical Informatics. Web site for the software Protégé, 2007. `http://protege.stanford.edu/` (accessed February 2, 2007).

[36] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure. *IEEE/ACM Transactions on Networking*, 12(2):205–218, Apr. 2004.

[37] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In R. Guerin, editor, *Proceedings of the ACM SIGCOMM 2001 Conference (SIGCOMM-01)*, volume 31, 4 of *Computer Communication Review*, pages 149–160, New York, Aug. 27–31 2001. ACM Press.

[38] V. Tamma, C. van Aart, T. Moyaux, S. Paurobally, B. Lithgow-Smith, and M. Wooldridge. An ontological framework for dynamic coordination. In *Proc. of 4th Int. Semantic Web Conf. (ISWC 2005)*, Galway (Ireland), 2005.

[39] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, Jan. 1997.

[40] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[41] F. von Martial. *Interactions among autonomous planning agents*. Elsevier Science Publishers B.V.:Amsterdam, The Netherlands, 1990.

[42] F. von Martial. *Coordinating Plans of Autonomous Agents*. Springer-Verlag New York, Inc., 1992.

[43] W3C. Simple object access protocol (SOAP), http://www.w3.org/tr/soap/.