A Programming System for Process Coordination in Virtual Organisations

Thesis by

Simon Woodman

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

University of Newcastle upon Tyne

Newcastle upon Tyne, UK

2008

(Submitted December 11, 2008)

To Sara

Acknowledgements

Firstly, I must thank my supervisor Santosh Shrivastava for his enlightening conversations, insight, support and funding throughout my PhD. Without his generosity this thesis would not have been possible. Similarly, Graham Morgan initially persuaded me to consider a PhD and without his foresight, I would not have started, let alone completed this thesis. There are many other people within the School of Computing Science who also deserve considerable acknowledgement for the part they have played during my PhD: John Lloyd and Pete Lee for providing me with the environment and support necessary to generate the ideas within this thesis; Paul Watson for allowing me work part time on a project that really deserved my full attention; Stuart Wheater for proving technical knowledge and expertise particularly in the early years and Savas Parastatidis for showing me that thinking differently and going against the flow is not necessarily a bad thing.

My parents have always put my education first and everything else second. For that I am eternally grateful and hope that I will be able to give the same privilege to my chidren. My Mum deserves more praise than I can lay down in a few short words. A kinder person one could not hope to meet. She has a mother's instinct on when to nag, when to console and when to celebrate. I could not ask for any more.

Einar Vollset has always been there to bounce ideas off no matter how idiotic they appear at the time. The evenings spent discussing our ideas, work and the generally putting the world to rights will not be forgotten. One never knows, eventually 'Bubble Search' may be vying for the Best Paper award. Olly Shaw, Dan Owen, Chris Smith and Paul Robinson have also contributed to my sanity: the former three in showing me that a PhD can be combined with sport (golf, squash and cricket respectively) the latter for providing helpful support, suggestions and giving me time to contemplate ideas whilst waiting for him to arrive.

Finally, I must thank my fiance, Sara. When we met she believed my thesis was 'almost finished'. We've been together for over three years and I'm now glad to say it is now, really, almost finished. For her patience with the early mornings and the late nights, for her understanding and most of all for just being herself I am once again, eternally grateful.

Abstract

Distributed business applications are increasingly being constructed by composing them from services provided by various online businesses. Typically, this leads to trading partners coming together to form virtual organizations (VOs). Each member of a VO maintains their autonomy, except with respect to their agreed goals. The structure of the Virtual Organisation may contain one dominant organisation who dictates the method of achieving the goals or the members may be considered peers of equal importance. The goals of VOs can be defined by the shared global business processes they contain. To be able to execute these business processes, VOs require a flexible enactment model as there may be no single 'owner' of the business process and therefore no natural place to enact the business processes. One solution is centralised enactment using a trusted third party, but in some cases this may not be acceptable (for instance because of security reasons). This thesis will present a programming system that allows centralised as well as distributed enactment where each organisation enacts part of the business process. To achieve distributed enactment we must address the problem of specifying the business process in a manner that is amenable to distribution. The first contribution of this thesis is the presentation of the Task Model, a set of languages and notations for describing workflows that can be enacted in a centralised or decentralised manner. The business processes that we specify will coordinate the services that each organisation owns. The second contribution of this thesis is the presentation of a method of describing the observable behaviour of these services. The language we present, SSDL, provides a flexible and extensible way of describing the messaging behaviour of Web Services. We present a method for checking that a set of services described in SSDL are compatible with each other and also that a workflow interacts with a service in the desired manner. The final contribution of this thesis is the presentation of an abstract architecture and prototype implementation of a decentralised workflow engine. The prototype is able to enact workflows described in the Task Model notation in either a centralised or decentralised scenario.

Declaration

All work contained within this thesis represents the original contribution of the author. Much of the material in this thesis has been published in conference proceedings or as book chapters as listed below. The material in Chapter Three has been published in 1 and the material in Chapter Four has been published in 2 and 3. The material in Chapter Five has been published in 4.

1. S.J.Woodman, D.J.Palmer, S.K.Shrivastava and S.M.Wheater. Notations for the Specification and Verification of Composite Web Services. *8th IEEE International Enterprise Distributed Object Computing Conference (EDOC '04)* , September 20-24,2004, Monterey, California pp. 35-46, IEEE Computer Society, 2004.

2. S.J.Woodman, S.Parastatidis and J.Webber, Protocol-Based Integration Using SSDL and π-calculus. In *Workflows for E-science: Scientific Workflows for Grids.*, I.J.Taylor, E.Deelman, D.B.Gannon and M.Shields (eds.), pp. 227-243, Springer-Verlag, ISBN: 978-1-84628-519-6, 2007

3. S.Parastatidis, S.J.Woodman, J.Webber, D.Kuo and P.Greenfield, Asynchronous Messaging between Web Services Using SSDL. *IEEE Internet Computing*, Volume 10, Issue 1, pp 26-39, IEEE Computer Society, 2006

4. S.J.Woodman, D.J.Palmer, S.K.Shrivastava and S.M.Wheater. DECS: A System for Decentralised Coordination of Web Services. In *Proceedings of Middleware for Web Services (MWS) 2005 Workshop held at EDOC 2005 Conference*, V.Tosic, A.van Moorsel, and R.Wong (eds), pp 24-32, IEEE, 2005

Contents

Chapter 1

Introduction

Organisations can be characterised by the business processes that they implement. Indeed the business processes that belong to an organisation have long been considered valuable intellectual property (IP) and much time and money is spent analysing and refining them in order to reduce the running costs of the business.

Although business processes are valuable IP for an organisation, they are not directly executable. They capture what steps a business takes to achieve a goal in abstract terms that are often technology agnostic. For example, one step in a business process might be 'send the insurance claim to the payment department'. This description does not dictate what technology should be used to achieve the goal. The claim could be sent via the internal mail, fax, e-mail, publish-subscribe messaging system or one of many other options. There are many technologies available for representing business processes in an executable format but workflow is one of the most popular. The main advantage of workflow technology for representing business processes is that there is usually a graphical representation of the workflow that has certain semantics and can be related to the business process. The graphical representation means it is easier for people who are not programmers to design, implement and monitor the workflow. It is easier to provide business level abstractions when leveraging workflow technologies than when using other systems such as developing a bespoke application. In addition it is possible for the environment which executes the workflow to provide services and non-functional properties such as fault tolerance, security and provenance. Many organisations, both physical and Virtual make use of these technologies to enact their business processes.

A Virtual Organisation (VO) is a temporary alliance of organisations that come together to share skills or core competencies and resources in order to better respond to business opportunities[1]. With the advent of such collaborations challenges are presented in describing and coordinating the work to be achieved by the Virtual Organisation. There are many different forms that the technical architecture of a Virtual Organisation may take depending on data, networking, security and other such considerations. For instance, some Virtual Organisations may employ a Trusted Third Party

[1] Adapted from [1] and discussed in Chapter 2.

(TTP) to host services and enact workflows on behalf of the VO. Other organisations may host secured services themselves and allow other members of the VO to use them. In some cases neither of the scenarios presented above may be acceptable to a VO. In such cases the VO may choose to host their own services and use distributed enactment of the workflow. Hence there is no 'one size fits all' solution to VO workflow enactment: any system must provide enough flexibility to allow each VO to choose the style of enactment that suits their requirements. Flexibility must be provided during the lifecycle of a workflow too. It is likely that the business processes will evolve over time and it must be possible to adapt the workflow in order to respond to these changes.

Web Services are applications which are accessible over a computer network and typically exchange data in eXtensible Markup Language (XML) format. We will describe Web Services and the technologies that they utilise in Chapter 2 but they provide a good basis for performing application integration in heterogeneous environments. Such an environment will be present in any single organisation, let alone a set of autonomous organisations. Any workflow system aimed at facilitating the execution of Virtual Organisations should be able to interoperate with Web Services.

As workflows become more complex it is increasingly likely that errors will be introduced. Such errors might include unreachable tasks or deadlocks when the execution progresses along certain paths within the workflow. It is desirable to be able to analyse a workflow and ensure freedom of such situations.

It is possible to distill the descriptions above into set of requirements that a solution for enactment of workflows within a VO must address:

1. Allow flexibility in the enactment model
2. Be able to interoperate with heterogeneous applications
3. Be amenable to verification of safety and liveness properties

One of the contributions of this thesis is the presentation of the *task model* as a solution to the above requirements. This includes various representations of the task model (graphical, XML based and π-calculus) and the respective mappings between them. The task model itself is not completely novel; the model presented here is based on the OpenFLOW task model [2] but it has been extended with other types of task to align it with Web Services and a formal model has been developed to allow verification of workflows described in this notation.

A complimentary facet of Virtual Organisation management and execution is the provision of descriptions and constraints on service usage. The workflows encoded in the Task Model notation will make use of application level services provided by the member organisations. Indeed it is the integration of these services that forms the basis of the VO operation. It is likely that these services will have complex interaction patterns and constraints on their usage to maintain consistency within

the organisation. Existing service description languages are not suited to capturing these interaction patterns. However, if such interaction patterns can be captured it would be possible to ascertain that workflows created to enact the business processes of the VO adhere to the required interaction pattern. Furthermore, it would be feasible to verify that a number of services are compatible with each other regardless of their implementation details.

The SOAP Service Description Language (SSDL) is a SOAP-centric contract description language for Web Services [3]. SSDL provides the base concepts on top of which frameworks for describing protocols are built. Such protocol frameworks can capture a range of interaction patterns from simple request-response message exchange patterns to entire multi service workflows within a composite application.

A "service" has become the contemporary abstraction around which modern distributed applications are designed and built. A service represents a piece of functionality that is exposed on the network. The "message" abstraction is used to create interaction patterns or *protocols* to represent the messaging behavior of a service. In the Web services domain, SOAP is the preferred model for encoding, transferring, and processing such messages. SSDL does not dictate the level of granularity of services or their visibility. It is possible to describe protocols that are intended to aid application integration within an organisation. Equally, it is possible to provide protocols to explicitly state the observable behaviour of a service provided by an organisation or set of organisations.

We will briefly introduce the main features of SSDL and its supported protocol frameworks. The second contribution of this thesis is the presentation of the Sequencing Constraints (SC) SSDL protocol framework for capturing the messaging behavior of Web services acting as part of a composite application or multiparty workflow. The SC SSDL protocol framework can be used to describe multi service, multi message exchange protocols using notations based on the π-calculus. By building on a formal model, we can make assertions about certain properties (e.g. lack of starvation, out-of-order messages, etc.) of interactions involving multiple Web services. We will demonstrate this notion of protocol compatibility and also show the relationship between the Sequencing Constraints and the Task Model presented in the previous chapter. Further, we will show how it is possible to verify that a workflow defined in the Task Model adheres to the protocols defined on the constituent services.

A programming system would not be complete without a method of enacting those programs. As the Task Model is aimed at providing a programming system for Web Services, specifically those that model business processes in Virtual Organisations, the provision of an engine to enact the workflows is a necessity. The third contribution of this thesis is the presentation of the architecture for a distributed enactment engine and a realisation of this architecture in our prototype workflow engine, DECS. When designing the Task Model notation a great deal of care was taken to ensure that the language did not unnecessarily restrict the architecture that could be used to enact it. Virtual Organisations may take many different architectural forms and there is no 'one size fits all'

solution. The same can be said for architectures of enacting workflows, flexibility is required to allow each Virtual Organisation to choose the style that suits their requirements.

In summary this thesis presents a number of contributions: firstly, a model for describing business processes which does not place constraints on a particular underlying enactment model. We show that the model is as expressive as contemporary languages for expressing business processes and is amenable to formal analysis to ensure that certain properties such as safety and liveness hold. Secondly, we present a method of describing the interaction patterns that a service adheres to. We show that it is possible to ascertain whether or not two services are 'compatible' and whether or not a business process respects the interaction patterns of each service it uses. Finally, we present the architecture of a platform to enact the distributed business processes that our model is able to define. We describe both an abstract version and present a concrete prototype implementation.

The remainder of this thesis is organised as follows. In Chapter 2 we present background information that is relevant to the subsequent chapters. We cover the theory and structure of Virtual Organisations before progressing onto a technical introduction to Service Oriented Architectures, Web Services and Workflow. We also describe the current state of the art in each of these fields and give some background information on formal notations. Chapter 3 presents the Task Model in various forms and uses pattern based analysis to compare it to similar languages. A formal model of the language is then presented and illustrated through the use of an example. Chapter 4 presents SSDL and its main features, concentrating in particular on the Sequencing Constraints protocol framework. Again, we show the formal backing and explain the relationship with the work presented in Chapter 3. In Chapter 5 we start by presenting the abstract architecture of a platform for enacting the business processes described in Chapter 3. Following on from this we describe a prototype implementation that has been undertaken. Finally, we draw our conclusions in Chapter 6 and describe the extensions to our work.

Chapter 2

Background

The background material relating to this chapter can be divided into a number of distinct sections. Firstly we will deal with the concept of Virtual Organisations which is the arena that the work presented here is intended to benefit. Secondly we will discuss Service Oriented Architectures with a focus on Web Services and methods of specifying the observable behaviour of those services. Leading on from Web Services we will cover workflow systems, specifically those that are aimed at providing inter-organisational, or distributed, workflow. Finally we will give a brief overview of process algebra which is used within this thesis to formalise certain aspects.

2.1 Virtual Organisations

The term Virtual Organisation was born during the 1990s in management literature as the fusion of technological advances and a socio-economic paradigm shift towards less rigid business relationships [4, 5]. Jan Hopland of Digital Equipment Corporation is attributed with the first use of the term whilst researching strategic management changes:

> It was clear we were entering into an age in which organisations would spring up overnight and would have to form and reform relationships to survive... 'virtual' had the technology metaphor. It was real and it wasn't quite real...it derives from the early days of computing when the term 'virtual memory' describe a way of making a computer act as if it had more storage capacity than it really possessed [6].

Kraft and Truex provide a list of the management terms which have been used to describe such organisations: virtual enterprise, dissipative organisation, imaginary organisation, adaptive organisation, learning organisation, flex firm, agile enterprise, pulsating organisation, network organisation and post-modern organisation [7]. Others have proposed and used other terms: modular organisation, value-adding partnership and organic network [8, 9, 10]

Camarinha-Matos and Afsarmanesh offer a taxonomy of virtual business entities which they group under the term "Collaborative Networks" [1]. It is easy to see where the term "collaborative networks" comes from: collaboration is required between organisations to agree and achieve the goal; and computer networks are utilised to achieve this collaboration. Specifically, they define Virtual Enterprises and Virtual Organisations:

Virtual Enterprise (VE) a temporary alliance of enterprises that come together to share skills or core competencies and resources in order to better respond to business opportunities, and whose cooperation is supported by computer networks.

Virtual Organisation (VO) a concept similar to a VE, comprising a set of (legally) independent organisations that share resources and skills to achieve its mission/goal, but that is not limited to an alliance of for profit enterprises. A VE is therefore, a particular case of VO.

The authors also distill definitions for other classifications of Collaborative Networks such as Dynamic Virtual Organisations (typically short lived VOs), Extended Enterprises (where there is one dominant participant) and Professional Virtual Communities (typically information sharing amongst individuals rather than organisations). Most of these are specialised versions of a Virtual Organisation and as such this thesis will not target them specifically. Instead we will consider the term Virtual Organisation to encompass them and if any differences are relevant they will be highlighted in the relevant sections.

Other authors give alternative definitions of a Virtual Organisation [11, 12, 13, 14, 1, 15], but most definitions share some common characteristics. Firstly there is no new legal entity defining the Virtual Organisation. Contracts may (and almost certainly will) define the relationships between member organisations, but there is no new organisation incorporated by them. Secondly, there is a common goal agreed on by the members. Each member will in addition have local goals but all will agree on the global goals of the Virtual Organisation. Opinion differs on whether competing organisations are allowed within a Virtual Organisation. For instance, can a VO contain a number of providers of the same product who bid to supply it to the other partners? It should be noted that even in this case each supplier agrees on the global goals of the Virtual Organisation. Thirdly, most authors agree that there is an extensive use of Information and Communication Technology (ICT) in the formation, execution and management of the VO. Finally, most agree that the aim of a Virtual Organisation is to allow each member to focus on its core competencies whilst exploiting markets more effectively and potentially opening markets that were inaccessible before.

Sanchez et. al. have analysed Virtual Organisations to see whether successful ones have certain properties [15]. They concentrate on Virtual Organisations which form from Virtual Breeding Environments [16] which are discussed in Section 2.1.2. The report categorises the VOs depending on the the 'level of technology' used by the VO ranging from 'high tech cases' to 'low tech cases'.

For instance, a bio-technology VO would reside within the 'high tech' grouping but a VO in the Chilean mining industry would class as a 'low tech' VO. In the VOs categorised as 'high tech' there is typically a close link between research and industry as well as a large amount of information sharing. In medium and low tech VOs there is more of an emphasis on planning and horizontal cooperation across a market. Two of the factors that appear to influence the success of a VO are Operational Support and Flexibility. Operational support is not necessarily ICT based, but will almost certainly be for the high tech VOs. It is worth noting the the level of ICT use was not found to be an influencing factor on success of VOs in another study [17]. However, Gruber and Noester concentrated on VOs residing in the manufacturing sector whereas Sanchez et. al. do not focus on one industry in particular. Flexibility is also identified by [11] as a key requirement for the success of a Virtual Organisation. Flexibility is seen as the power to respond to changing market conditions more rapidly than competitors and thus obtain a market advantage both at formation time and during execution. An area addressed in this thesis is flexibility of business process enactment which will be described in Chapter 5.

Becoming part of a Virtual Organisation has both benefits and costs [18]. The costs of joining a Virtual Organisation are mainly concerned with aligning the member organisation's software infrastructure and business processes with those necessary for the VO to operate. The advent of standardised Web Services as discussed later in this chapter has significantly reduced some of these costs but not others. Other costs that need to be taken into account are the amount of retraining required, the preparation of information required by the VO and whether or not specialised certification is required [19]. The benefits of joining a VO will vary depending on the specific aims of the organisation but may include the creation of new marketing channels; the utilisation of otherwise unused resources [5]; political or social advantages (for instance increasing the status of the organisation). Furthermore if the business processes operated by the organisation required altering (a cost) the result may be more efficient and potentially allow activities that were not possible before.

2.1.1 Structure of Virtual Organisations

The structure of Virtual Organisations can vary widely and there does not seem to be a structure that indicates likelihood of success [17]. However, it is possible to define a number of 'dimensions' in which Virtual Organisations may be classified [4]:

Risk and Reward: ranging from a joint venture with shared risk and reward to client-contractor relationships.

Life-span: ranging from a permanent existence but based on remote relationships to a limited, single project.

Membership: specific aspects (resources, people, services) of member organisations to a joint venture of individual contractors.

The authors of [12] attempt to create a taxonomy of the structure of Virtual Organisations by analysing information flows. They concentrate on three types of information: firstly, data related to the *planning* and formation of the VO. This includes the creation and integration of business processes. Secondly, data related to the *operation* of the VO, namely the data that must be exchanged in order for the goal of the VO to be realised. Finally they consider data relating to the *coordination* of the VO such as state updates and other information related to control flow.

The Star Alliance as shown in Figure 2.1 is a grouping of independent organisations where a core organisation takes a leading role. The core organisation may own key knowledge or resources and may be significantly larger than the other organisations. It is the core organisation that presents the face of the VO to the customers.

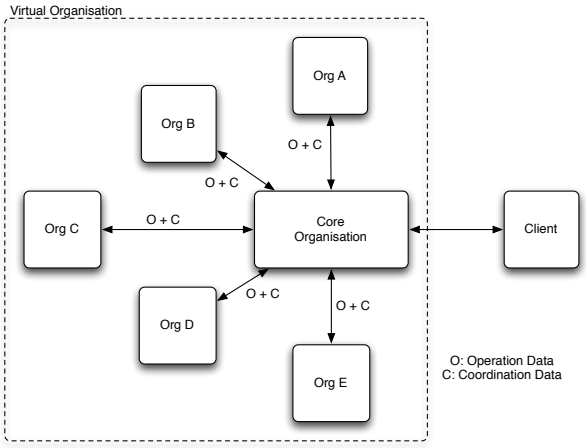


Figure 2.1: Star Alliance

One example of a Star Alliance is when an organisation outsources operations that it does not consider to be within its core competencies. The planning is done by the core organisation and the satellite organisations will only communicate with the core organisation, not each other. It is possible that an organisation participates in a number of star alliances - a satellite member of one Star Alliance may be the core member of another. It is not possible to remove the core organisation from a Star Alliance although it may be possible to replace satellite organisations.

A Market Alliance, shown in Figure 2.2, is a specialised case of a Star Alliance where the core organisation is only responsible for sales and marketing functions. As with the Star Alliance, the core organisation is responsible for all contact with the customer, planning and in this case coordination too. A Market Alliance is typically used as an online marketplace where producers can aggregate their products or services with other similar organisations. Market Alliances may be long-lived and it is feasible for the satellite organisations to leave (and for others to join) without affecting the

alliance (assuming that there is still a sufficient number to make the alliance viable). However, it is not feasible for the core organisation to leave the alliance - this would result in the alliance disbanding.

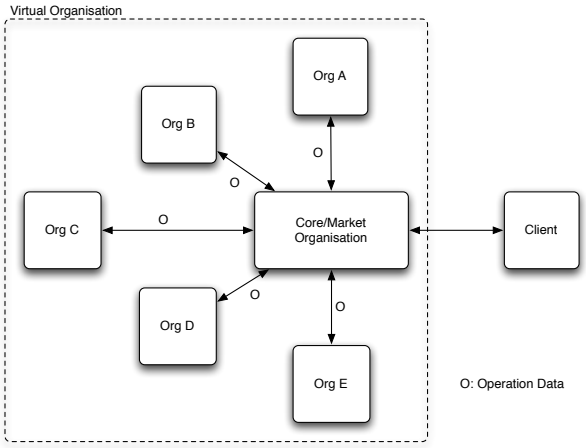


Figure 2.2: Market Alliance

A Co-Alliance (Figure 2.3) exists when each member organisation has equal commitment and stake in the Virtual Organisation. They communicate with each other bi-laterally and also with the customer throughout their existence which may be short or long-lived. The membership of a Co-Alliance is likely to be fixed although under some circumstances certain organisations may be replaceable with others that perform *exactly* the same function. It is not typically possible for an organisation to leave a Co-Alliance.

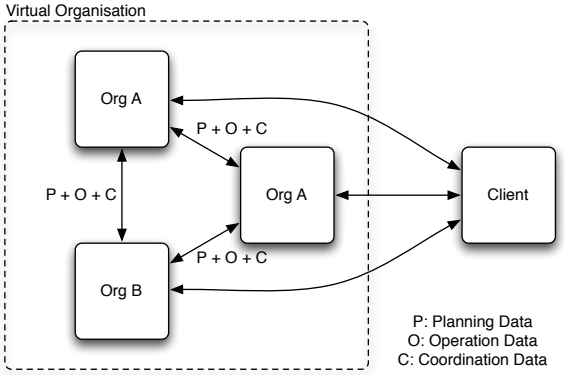


Figure 2.3: Co-Alliance

One of the principle attributes of a Co-Alliance is that the members have an approximately equal stake in the alliance - they are peers. There is no organisation that is significantly 'stronger' in the Alliance than others; if such an organisation exists a Star Alliance has been formed. The formation of a Co-Alliance will involve planning an agreement from all the members on the goal and method

of achieving it. This could take the form of a shared workflow[20], choreography description[21], electronic contract[22]or service level agreement.

A Value Alliance, shown in Figure 2.4, differs from the Virtual Organisations presented previously in that it is based on the value (or supply) chain. Each member adds value to the work done by the previous organisation in the chain. Value Alliances typically reflect the structure of certain manufacturing industries and often deal with physical goods rather than 'knowledge' or information services.

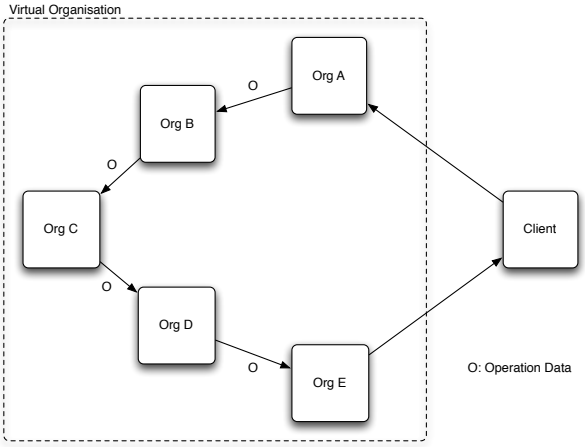


Figure 2.4: Value Alliance

Value Alliances may be circular, in that the first and last organisation in the chain may be the same organisation. The customer will only deal with the first and the last organisation and there is no requirement that the customer who initiated the request is the one receiving the goods. An example of a value chain is a customer ordering flowers online to be delivered to someone else. The flowers are ordered from Organisation A who then enlists Organisation B (and possibly subsequently Organisation C too) to deliver the flowers to the recipient. Each organisation has added value, in this case, moving the flowers closer to the customer. Again, in a Value Alliance there is no core organisation and planning is usually done beforehand between individual organisations that are neighbours in the chain.

A Parallel Alliance (Figure 2.5) models the case in a supply chain when one value-adding organisation must coordinate closely with another. As the organisations are independent it is not possible to integrate them and so their operation must be linked. An example of a Parallel Alliance is when one organisation is designing a product and another is constructing it. The organisations are clearly distinct but their functions need to be linked - the designer must not conceive something that cannot be built economically.

The authors consider a sixth type of Virtual Organisation, namely the Virtual Face. This is the simplest type of Virtual Organisation and is simply the online presence of a physical organisation.

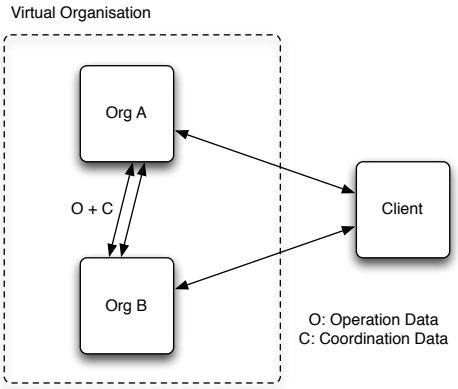


Figure 2.5: Parallel Alliance

It is included in the taxonomy for completeness but does not constitute a Virtual Organisation in the context of this thesis as there is only one member simply trading through a different channel.

It should be noted that the above classifications of Virtual Organisations are simple, elemental structures. In practise the structure of a Virtual Organisation is likely to be a combination of the different structures. In addition, each organisation may be a member of a number of different Virtual Organisations irrespective of whether it is a core member or satellite member.

2.1.2 Life-cycles of Virtual Organisations

There are many ways in which a Virtual Organisation may be formed, with different methods suiting the different structures of Virtual Organisations identified in the previous section. The formation of Star Alliances, especially those which deal with outsourcing, is likely to be driven by the core organisation. The core organisation will deal with selection of satellite organisations, will dictate technologies to be used and standards to be followed. It is possible that the core organisation may hold a reverse auction [23] in order to select suppliers who are able to fulfil their requirements.

The concept of a Virtual Breeding Environment (VBE) is introduced in [1] and represents an association (also known as a cluster or pool) of organisations that have the "potential and will" to co-operate with each other. The aim is that when one member identifies an opportunity it is easier to select a set of organisations (a subset of the VBE) to form a VO with. It is possible that the members of the VBE have agreed on certain common technologies to allow easier collaboration or that they belong to a common vertical market sector. In fact VBEs themselves are one type of Virtual Organisation even though they typically lie dormant and have little or no collaboration until an opportunity is identified.

A conceptual model of Virtual Organisations is presented in [24] which consists of a number of sub models represented in UML. The models include ways of representing business service processes which are provided by VOs, virtual and physical production networks (management aspects of

collaboration) and virtual business services. The models of the business process can be refined but are not executable. In [25] the authors indicate how business processes expressed in such models relate to service-oriented computing and could be enacted using the Business Process Execution Language (BPEL4WS).

The GOLD Project [26] has identified a number of services that are required during the execution of the Virtual Organisation. Security issues such as authentication and authorisation must be addressed. They advocate the use of federated identification to establish and verify the identity of users, agents, services and components [27, 28]. Central to collaborative working is a method of notifying (interested and authorised) parties to particular events. Their research suggests that a peer-to-peer notification service is beneficial to allow members to coordinate the work they are carrying out. Workflow technologies are used by each participant to carry out the work required and notify the other members (through the notification service) when this has been achieved. In some cases it is necessary to monitor and mediate the interactions between parties to ensure they correspond to some pre-arranged pattern. In such cases each member of the VO must be accountable for their own actions; must comply with any contracts covering the VO; and must ensure that their own legitimate actions are recognised. These are achieved using 'contract-mediated interaction' and non-repudiation protocols [29, 30]. The final aspect of VO enactment identified was the need to be able to store and manage information and in some cases determine how this information was derived.

There is very little information in the literature dealing with the termination of Virtual Organisations. Once the opportunity they were created to exploit is no longer viable the VO will need to be disbanded. Owing to the lack of a new legal entity being formed by the creation it is possible that the member organisations simply cease to cooperate and concentrate on other business goals. In this case it will be necessary to perform tasks such as revoking any access rights, reclaiming equipment for other use and archiving data etc. It is equally possible in some situations that the VO is left dormant for months or even years. If the VO is providing a service that no-one is using but it is not costing the member organisations significant amounts of money to provide then it might cost more to dismantle the VO than to leave it in a 'running' state. It is likely at some point, perhaps whilst auditing or re-engineering their business processes the cost/benefit equation may change and the member may leave the VO. It is also possible at the dissolution of the VO that each partner 'rates' the other partners in a similar way to the feedback mechanism on ebay [31]. This rating can be fed back into any VBE or similar consortium for use in future vendor selection [32].

2.1.3 Business to Business Messaging

A number of standards exist which define the syntax and semantics of sets of messages which one organisation may send to another. Such standards typically define dictionaries of standard terms, the structure of messages including any compulsory and optional sections and the way that such

messages may be composed into protocols that are meaningful at a business level. Examples of such standards are ebXML[33] and the RosettaNet suite of standards[34]. The following Section will take a look at the latter in a little more detail.

2.1.3.1 RosettaNet

RosettaNet [34] is a consortium of over 400 of the worlds leading IT, semi-conductor manufacturing, electrical component and solution provider companies. The aim is to define and promote open e-business standards to enable and lower the cost of inter-organisation interaction. RosettaNet define reusable protocol specifications that achieve a small business goal. The protocol specifications are termed PIPs (Partner Interface Process) and achieve goals such as submitting a purchase order, requesting the price and availability of goods or sending remittance advice. The definition of each PIP consists of a number of items: schema for the messages exchanged; message sequence charts indicating the ordering of message exchange; quality of service requirements such as timeouts, security considerations, non-repudiation and retries. [35]

In addition to PIPs, RosettaNet also defines the RosettaNet Implementation Framework (RNIF) which is the specification of a middleware framework that supports PIPs. The framework defines a number of common interaction patterns which form the basis of the message sequence charts mentioned above. Each PIP follows one of the following interaction patterns:

asynchronous single-action activity Partner A sends a Business Message to B. B sends a Receipt Acknowledgement or an exception to A. A timeout or exception triggers exception handling behaviour.

asynchronous double-action activity Partner A sends a Business Message to B who responds with a Receipt Acknowledgement and another Business Message to A. A in turn sends a Receipt Acknowledgement to B. Timeouts or exceptions instead of Receipt Acknowledgements trigger exception handling.

synchronous single-action/double-action activity By default PIPs are asynchronous but provision is made for synchronous transport but most of the quality of service characteristics are dropped. In the single action version Partner A sends a Business Message to B who may respond with a Receipt Acknowledgement or not at all. In the double action version B responds with another Business Message and no Receipt Acknowledgements are allowed.

A full description, including message sequence charts, specifics of the quality of service requirements can be found in [36]

The RosettaNet PIP standardisation effort began before Web Services became a popular way of achieving inter-organisational integration. Although the messages that are defined in each PIP

contain XML data it is not trivial to map this onto Web Services. One of the challenges is that the layers in the PIP definitions and the Web Service layers do not match directly. In the PIP definitions the business logic leaks into the messaging layer (the RNIF defines how to handle business faults and whether messages must be ordered or may arrive simultaneously). In addition, the messaging logic leaks into the business layer (the PIP description defines the number of retries). These facts make it non-trivial to map PIPs onto Web Services in an elegant manner. The approach taken in [37] is to map the complete messaging behaviour of a PIP into a BPEL process. Whilst this is viable it amplifies the problem of messaging behaviour leaking into the business process layer and significantly complicates the business processes. The author shows how it is possible to create templates for PIPs so that the implementation effort can be reduced and makes extensive use of the BPEL fault handler mechanisms for dealing with timeouts. Other authors have chained PIPs together to provide a description similar to a choreography [38]. They have an interesting approach to producing template workflows for each party through the use of projections but appear not to deal with failure states within a PIP.

Each PIP is intended to be an atomic unit of work that achieves a small business goal. A PIP can effectively complete in one of two states: success or failure. It should be noted that these two states are concerned with the execution of the PIP rather than any business implications that it might have. The success state indicates that all messages were received within the timeout requirements and validated against the required schemas. The failure state indicates that something went wrong during the execution. The RNIF specification states that if the PIP ends in a failure state another PIP should be initiated to deal with this failure (PIP0A1:Notification of Failure). The problem is worsened by the timeouts specified in the RNIF: it might be up to 24 hours before failure state is reached by which point one of the business processes might have progressed without realising the error. In [39] an approach to preventing this is outlined that synchronises the PIP execution at certain critical points in the business process.

2.2 Service Oriented Architecture

Service Oriented Architecture (SOA) is the term given to the architectural style that has recently become popular to allow the creation of large scale distributed software systems [40]. The fundamentals of SOA are not new and some claim that the name is unfortunate as discussions around it often progress out of the scope of architecture and into business design [41]. Much that is written about SOA has a focus on Web Services technology [42, 43, 44, 45] although this is not always beneficial; the fundamentals of SOA are technology agnostic and can be realised using other technologies than Web Services. OASIS, a global standardisation consortium, attempt to define a SOA in a technology neutral manner [40]:

> "Service Oriented Architecture (SOA) is a paradigm for organising and utilising distributed capabilities that may be under the control of different organisational domains."

SOA promotes the decomposition of applications into independent abstractions called services. Services are autonomous and provide capabilities to higher level services without the higher level service needing to know the implementation details of the capability. So far, SOA sounds remarkably similar to Object Orientation (OO) [46]. As services are unlikely to be within the same address space, or even administrative domain they have a lesser degree of coupling than is assumed within OO principles. Services must expose descriptions of how to interact with them in terms of message structure and ordering. It is the combination of services, messages and this meta data that form the basis of a SOA [44].

Many authors offer alternative definitions of SOA to that of OASIS which was presented above [47, 48, 49] but most agree with the "Four Tenets of Service Orientation" provided by Don Box in [50]:

Boundaries are explicit. In an SOA, services communicate through the exchange of messages across service boundaries, which are well-defined and explicit. Users of services have no knowledge about what is behind a boundary, which keeps service implementations private and decoupled from other services. Because services span separate processes, trust domains or geographical boundaries, each boundary crossing is potentially expensive in terms of processing overhead, performance penalties or complicated failure scenarios. For this reason, inter-service communication must be consciously distinguished from local method invocations. By making boundaries formal and explicit, developers recognise this difference between local and remote communication.

Services are autonomous. Services are self-governed and fully control the logic they encapsulate. They are modular building blocks that do not require knowledge of each others internal workings in order to interact. As a result, services can evolve independently from each other as long as they do not alter their public contracts. Moreover, as the topology of a service-oriented system is expected to change over time, adding, upgrading or removing services should not disrupt the overall system.

Services share schemas and contracts, not classes. Services maintain implementation independence by exposing only schemas and contracts that are expressed in a platform-neutral format. Schemas define the structure of messages a service can receive or send, while contracts determine the mechanics of these interactions. Together, schemas and contracts are shared beyond service boundaries.

Compatibility is determined based on policy. Besides using schemas and contracts for agree-

ing on structural compatibility in terms of messages and exchange sequences, services might have further constraints on the semantics required for communication to take place. Therefore, both requirements and capabilities are expressed in a machine-readable policy description. This separates the description of a service's behaviour from the specification of constraints for accessing it.

2.2.1 Loose Coupling

Using a Service Oriented Architecture attempts to produce a more loosely coupled system. The words 'loosely coupled' have no formal definition and many people have attempted to create a set of metrics to determine the level of coupling within a system [51, 52, 53, 54]. The common areas that reappear in each of the definitions are: the level of decomposition of the system and the granularity of modules; whether it is easy to compose these modules into a higher level application; whether the modules can be reused in multiple applications; whether the modules are independent of each other with respect to time and shared state. Interestingly, these are remarkably similar to the set of requirements that Meyer gave in order for a design is 'modular' [55].

Service Oriented Architectures aim to be more loosely coupled than large scale RPC or distributed object systems. Remote Procedure Calls (RPC), is a mature technology originating in the 1970s as RFC 707. The intention is to abstract away the network complexity and create an environment that is familiar to developers used to writing non-distributed applications. Later, in the 1990s, more complex systems such as CORBA [56] and DCOM [57] were developed. Following those, Java developed Remote Method Invocation (RMI) [58] and consequently Enterprise Java Beans [59] which all added functionality but had RPC as their underlying method of communication.

The way of providing 'location independence' in these systems is through the use of proxy and stub classes. These are local objects, that can be automatically generated and contain the code necessary to manage the remote object and invocations to it. The programmer uses the stub object in the same manner that they would if the object really were local. RPC systems were and still are successful and many applications make extensive use of RPC but some authors have pointed out that RPC is not ideal for Internet-wide computing [60, 61, 62, 63]. In [64], Waldo et. al. argue the diametric opposite of the aims of RPC:

> "objects that interact in a distributed system need to be dealt with in ways that are intrinsically different from objects that interact in a single address space"

The authors argue that the fundamental differences between local and remote objects are too great to be abstracted away. Specifically, they identify the problem areas as: network latency, memory address space, concurrency and partial failures. These areas must be explicitly handled and it is not possible to abstract them away.

When developing distributed applications that involve long running activities, the use of synchronous communications can be problematic. This means that the modules are not 'time-independent' of each other as one module must be online for the other to function. When a synchronous method invocation is made the client and server keep the channel open to carry the response. This places additional load, in terms of both complexity and performance, on the server which must maintain large numbers of concurrent connections. Further, the client and server are tightly coupled as the server must be available for the client to execute. In distributed environments such as the Internet when links might fail and applications become unavailable this is not desirable.

Messaging queueing systems, generally referred to as Message Oriented Middleware (MOM) attempt to resolve the problem of the client and server needing to be available at the same time, and result in systems that are more loosely coupled. In messaging systems there is not generally a technical distinction between the client and the server; the client can both send and receive messages, as can the server. The distinction is drawn for pragmatic reasons and ease of explanation. In messaging based systems it is possible to provide interfaces for different technologies overcoming the fact that most successful RPC based systems are based on homogeneous technology [63]. Additionally, it is possible for the message system to queue the messages so that if one of the applications is not available the message can be delivered at a later time. The final advantage of message based systems is that they allow the client to continue working whilst the server is processing the message. This allows higher throughput and is contrary to most RPC systems which are usually synchronous [60].

Although messaging systems are powerful and remove some of the problems found in RPC systems they are not without their own drawbacks. The asynchronous nature of the applications means that developers are generally less familiar with them. This results in higher cost of development and increased faults. There are also issues relating to message ordering and synchronisation which can cause bugs and race conditions. Such errors are difficult to find and might not manifest themselves until years into the applications life.

2.2.2 Web Services

Over the past few years, Web Services have been a hot topic for standardisation committees, researchers and industry alike [44, 65, 66, 67, 68, 53]. Rather than covering all of this work we will give an overview of the underlying concepts and then focus on the aspects most relevant to our work. The W3C identified characteristics common to most Web Service applications and the relationship between them [69]. In addition they define a common vocabulary for discussing Web Services and include a definition of what a Web Service is. However, instead of using their definition we will diverge slightly and provide the following definition which is more aligned with our view:

> A Web service is a software system designed to support interoperable machine-to-machine

> interaction over a network. It has an interface described in a machine-processable format. Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialisation in conjunction with other Web-related standards.[1]

Vogels states in [63] that Web Services can be viewed as XML processors that send and receive XML documents over a combination of transport and application protocols. This view is similar to that argued in [49] that services process messages using a combination of application logic and back end resources. As shown in Figure 2.6, this implies that a service consumes a message from the network, processes it and perhaps deposits another message onto the network (although this is dependant on the processing of the previous message). It should be noted that this is not dissimilar to the way that many organisations currently do business. A message is received, say a fax wishing to order some goods. The message (in this case a piece of paper) is routed to the order processing department who decides whether the order can be fulfilled. The customer who sent the initial message is then contacted to notify them of the decision, perhaps via fax, email or telephone (equivalent of putting a message back onto the network).

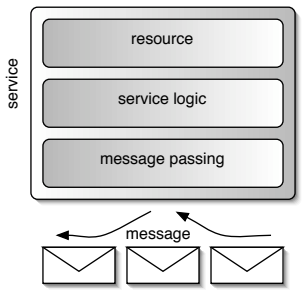


Figure 2.6: Web Services are message processors

The initial architecture for Web Services is founded on three complimentary specifications which are built using XML to achieve platform and language independence[2]. These specifications are Web Services Description Language (WSDL), SOAP[3] and Universal Description, Discovery and Integration (UDDI) [66, 53, 43]. These specifications allow for the well known 'publish, find, bind' paradigm as demonstrated in Figure 2.7. SOAP provides the semantics for communication, WSDL provides a machine readable format for specifying the behaviour and UDDI specifies how to publish and discover information about the services.

In the previous Sections we discussed the concepts of SOAs and loose coupling. Web Services are often described as a technology for realising loosely coupled SOAs as the Web Services frameworks

[1] The W3C definition mandates the use of WSDL as the interface description language which we do not feel is required.

[2] XML is known as the *lingua franca* of Web Services

[3] Initially, SOAP stood for 'Simple Object Access Protocol' but this somewhat misleading as it is neither simple nor object-oriented. In 2001 the W3C dropped the expansion, preferring simply 'SOAP'.

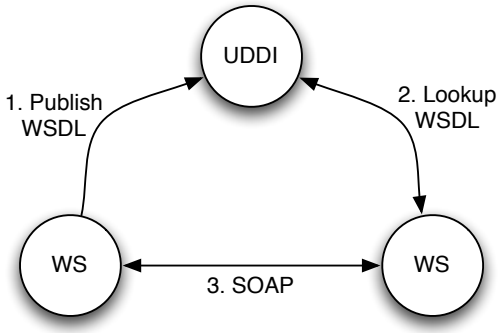


Figure 2.7: Initial Web Services Architecture

provide specifications and models that are aligned with concepts of SOAs. However, we must be careful as the use of Web Services alone does not imply the creation of a well architected SOA. For instance, wrapping existing applications in Web Services is a practise that is supported by many toolkits. However, this can lead to semantic mismatches, data-type mapping and state management issues and limitations in scalability and performance [70]. It is possible to realise a loosely coupled system using other technologies such as CORBA and DCOM or using Message Oriented Middleware. We have chosen to use Web Services due to their prevalence in application integration. They do not rely on homogeneous software platforms or low latency networks, nor do they suffer from the political or logistical problems of hosting message queues at particular destinations.

2.2.3 WS-*

One of the problems with the basic architecture suggested by publish-find-bind paradigm is that it does not cover more advanced topics such as security, reliable messaging, transactions or service orchestration [71]. For this reason the basic relationship presented above is commonly known as *first generation Web Services architecture* [43]. The more advanced topics have been covered by multiple additional specifications which are collectively known as WS-*. Figure 2.8 shows how some of the WS-* specifications relate to each other but for a more complete depiction (which is far too large to recreate here) the reader is referred to [72]. The number of specifications is large and not all have been implemented or gone through a recognised standardisation process. Also, there are some which co-exist in the same problem domain but have been authored by different organisations through different standards bodies. Vinoski discusses this problem in [73] and notes that no specification standardisation has yet occurred and the vast number of (often overlapping) specifications can be confusing [74, 75]. The problem is aggravated by the fact there are no clear guidelines for best practises and the last W3C architecture note predates many of the WS-* specifications. It is also not clear if there is a set of specifications that can be shown to cover the whole spectrum whilst not

contradicting each other.

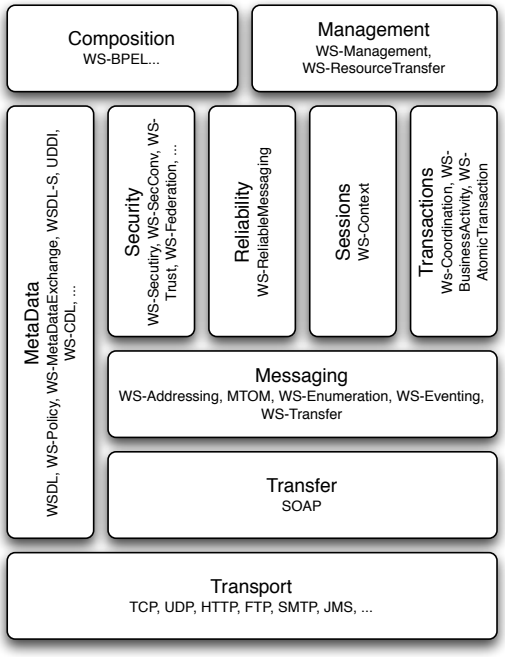


Figure 2.8: Overview of the set of specifications that are known as WS-*. Adapted from [71, 76]

2.2.4 SOAP

SOAP is an application protocol that allows Web Services to communicate in a standardised manner [77]. SOAP is transport agnostic and therefore can be layered on top of numerous transport protocols, the most common being HTTP, SMTP and JMS. SOAP messages are written using XML, allowing them to be interoperable across heterogeneous application platforms, self describing and both human and machine readable[78]. A SOAP message consists of an envelope which contains zero or more header elements and one or more body elements. The header elements are used for extensibility; the SOAP processing model allows as many headers to be defined as necessary and defines rules as to whether or not the individual headers must be processed. Many of the WS-* specifications make extensive use of SOAP headers to transmit information relating to security, reliability or transactional semantics. The body of the SOAP message contains the 'payload' of the message to be processed by the recipient.

2.2.4.1 WS-Addressing

WS-Addressing [79] is one of the key specifications within WS-*. Prior to its agreement there was no standard manner to embed information relating to addressing in a SOAP message. Instead,

```
<soap:Envelopexmlns:soap="http://www.w3.org/2001/12/soap-envelope"
            xmlns:a="http://www.w3.org/2005/08/addressing/">
  <soap:Header>
    <a:MessageID>uuid:6B29FC40-CA47-1067-B31D-00DD010662DA</a:MessageID>
  </soap:Header>
  <soap:Body>
    <purchaseOrder xmlns="urn:example:schemas">
    <item>chicken</item>
    <quantity>200</quantity>
    </purchaseOrder>
  </soap:Body>
</soap:Envelope>
```

Figure 2.9: Example SOAP Message

the transport level (e.g. HTTP) was used to define the destination of the message. However, as SOAP is inherently transport neutral, and a SOAP message may pass through many intermediaries and over different network transports this led to vendors using non-standard ways of specifying addressing properties. WS-Addressing addresses this requirement and in addition defines headers for correlating messages that are part of asynchronous interactions (e.g. *MessageID*, *RelatesTo* and *ReplyTo*). This last element also SOAP to model three way conversations where the recipient of the 'response' message is different to the service that sent the 'request'.

2.2.4.2 Interaction Styles

SOAP was initially intended to serialise and deserialise object graphs into an interoperable format (i.e. XML for sending over the network). As such it was a way of unifying proprietary RPC communications used by distributed object systems. From version 1.1 (released in 2000) the SOAP specification has included two distinct *styles* of messages: *RPC-style* and *document-style*. In addition, the way that the XML data is represented can be defined in two ways: by *encoding rules* (i.e. encoded) or by *external schemas* (i.e. literal). This leads to four different ways of representing the same data in a SOAP message, however, in practise, only *document/literal* and *RPC/encoded* are common (the latter is now outlawed by the the WS-I Basic Profile [80]). Confusion between these styles has caused problems for both developers of SOAP toolkits and users to whom SOAP appears more complex than need be [51].

In RPC-style SOAP messages the body of the message contains an element with the same name as the remote method to be invoked and elements containing the values of the parameters of that method. Similarly, the response message contains an element (usually the methodName*Response*) which contains the return values of the method. In most cases the interaction is synchronous, in request-response fashion, and use HTTP as its interaction model fits well with RPC. RPC style Web Services are still widely used even though the WS-I Basic Profile dictates the use of document/literal

as RPC style services are typically the easiest way to 'bolt on a Web Services interface' to an existing system. This involves using toolkits to generate WSDL from local objects and allows developers to continue using familiar programming abstractions [81]. However, services developed in this manner often exhibit the same problems that were identified in RPC systems in Section 2.2.1. That is, they are often tightly coupled, limited in scalability and allow method signatures to leak into the public contract [64]. This can make it hard for service implementations to evolve without breaking the public contract. Services created in this manner tend to be 'chatty', that is, they require many (small) messages to achieve a goal. This can increase overheads as the messages are being sent over a network and it does not fit naturally with business processes which usually operate at a higher level of abstraction.

Loughran and Smith discuss further problems with developing RPC-style Web Services, namely the Object-to-XML mapping issue [78]. The problem is that there is an impedance mismatch between the representational capability of XML Schema [82] and the type systems of most Web Service implementations. In most cases XML schema is a richer language for describing types than object-oriented languages such as Java or C#. This means it is not possible, in the generic case, to serialise method and return parameters into XML and vice versa.

In Document-style interactions, the body of the SOAP message encapsulates a complete XML document whose structure is normally defined in XML schema. When considering Web Services that involve business processes and business documents this offers a more natural fit than RPC-style. The body of the SOAP message does not contain references to method names or parameters. Document based messages tend to be larger than RPC ones and contain all the contextual information necessary to process that message. The fact that the messages are larger and are focused around business documents tends to result in less 'chatty' services where fewer messages are required to achieve a goal. The problem with using Document-style messaging is that many of the toolkits available focus on the former, RPC-style interactions, and these offer abstractions that developers are familiar with. More familiarity with XML technologies and asynchronous programming styles is required for developers to move to Document-style asynchronous interactions. When services are developed in this way they are more likely to function as independent, loosely coupled components as appropriate for Service Oriented Architectures [83, 60, 51, 78, 45, 81, 84]

2.2.5 WSDL

Web Services Description Language (WSDL) is the common way to define the public contract of a Web Service [85]. A WSDL document is written in XML and contains two parts: an abstract part and a concrete part. The abstract part allows the author to define types using XML Schema which are used to specify messages that are consumed or emitted by the service. These messages are combined into logical groups of operations. The concrete part binds the abstract part to a particular network

location and transport protocol. The advantage of the separation between abstract and concrete is the reuse of the abstract elements to allow different transport protocols (eg. HTTP, SMTP or JMS).

In its current version, the WSDL 2.0 Core specification is over 100 pages long an contains many improvements over the previous version. However, it has not universally found favour, and many current Web Service toolkits still use WSDL 1.1[86, 87, 88]. The common complaint is over the unnecessary weight and complexity of the specification and the fact it is unable to capture all but the simplest of interaction patterns. WSDL only really addresses the connectivity issues of message format and transport binding. It is argued however, that the complexity in distributed systems does not lie in connectivity but with interoperability such as message ordering, timing and QoS [89].

WSDL defines eight simple message exchange patterns (MEPs) that are used to group messages into operations. These are shown in Figure 2.10 with the most complex containing two messages where one of them might be replaced by a fault message. There are four MEPs that contain one message, two of which might have an optional second fault message, and four that contain two messages, two of which might have the second message replaced by a fault message. The most complex 'protocol' that WSDL is able to describe precisely is that either message B or a fault message will follow the receipt of message A. It is not possible to describe any higher level ordering between the MEPs themselves, only the ordering within the MEP.

It seems that WSDL's design, in a similar way to SOAP, is heavily based on RPC ideas even though, strictly speaking it makes no assumptions about the synchrony. The main abstraction is an *operation* which has led developers alike to use it as an Interface Definition Language (IDL) [90]. This is not helped by vendors who provide toolkits which will generate stubs to allow location transparency, a practise which is not ideal for a technology aimed at crossing organisational boundaries. Indeed in [63], Vogels points out that Web Services used in this manner might solve some interoperability issues but "provide no magic that can suddenly overcome what excellent protocol architects were unable to achieve with established RPC systems."

Owing to design decisions, WSDL is unable to place any constraints on message ordering apart from trivial cases within a MEP. Making message ordering explicit in service descriptions is something that is necessary to reduce the complexity of distributed systems [89]. Although it is possible to layer other specifications on top of WSDL, for instance abstract BPEL or WS-CDL this can further increase the complexity of the Web Service description. The protocol frameworks specified in SSDL (including the one presented in Chapter 4) allow the explicit specification of message ordering without requiring additional complexity or layered specifications.

A further design decision which increases the complexity of WSDL is that it is able to describe Web Services which communicate using other application protocols than SOAP. This is even though the description given by the W3C in Section 2.2.2 explicitly names SOAP as the application protocol that Web Services use. Whilst it is noble to make WSDL as generic as possible this means that

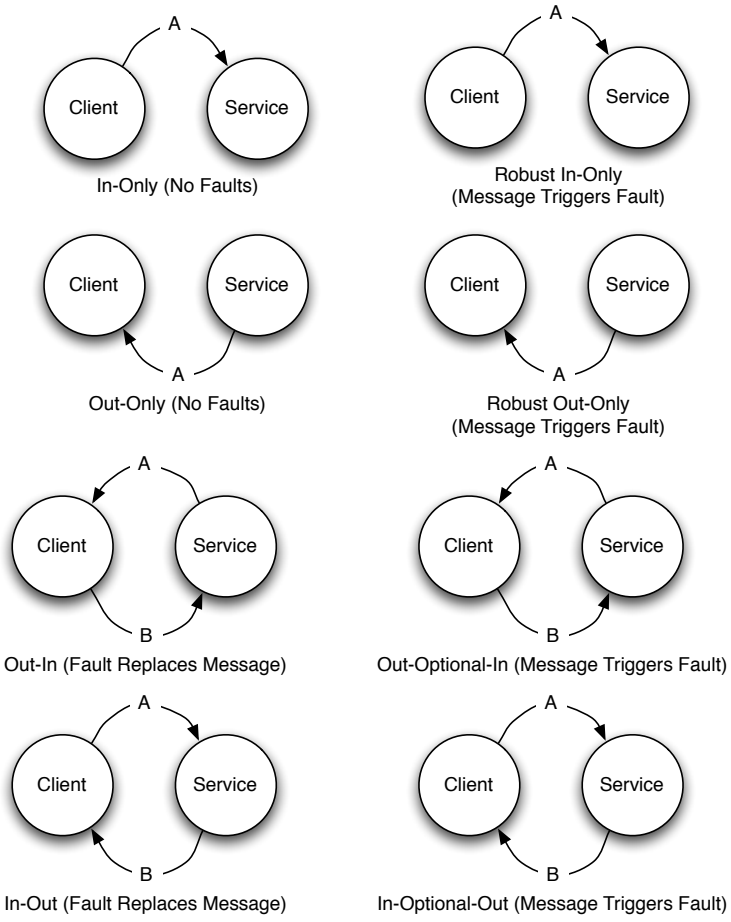


Figure 2.10: Message Exchange Patterns in WSDL 2.0

users have to explicitly define SOAP transports, and SOAP interaction styles/encoding rules, for each of their services adding unnecessary verbosity to the description. SSDL [91] was created as an alternative to WSDL that assumes SOAP is the application protocol used and thus reduces verbosity of both the specification document and Web Service descriptions written in it.

2.2.6 Interaction Protocols

Interaction protocols describe the relative ordering of messages that a service exchanges [4]. They are documents that advertise the interactions with other services but do not consider how the service is implemented [92].

There are two commonly used perspectives that interaction protocols take, shown in Figures 2.11 and 2.12 which differ in who's viewpoint is used to describe the protocol. Firstly it is possible to describe it from the perspective of the service advertising the protocol in which case we term it 'service centric'. Secondly it is possible to describe the protocol from the perspective of a global, omnipotent observer leading to the 'global perspective'. Sometimes the service centric approach is termed the 'local view' or 'behavioural interface' and the global perspective is also known as a choreography. In this thesis we use these terms interchangeably. Each of the perspectives has its advantages and disadvantages: service centric protocols are convenient for the service provider and are a natural extension of standard interface descriptions which developers are familiar with. However, service centric protocols must be combined with each other to be able to study the compatibility between services, something that the global perspective gives as standard. One disadvantage of the global perspective is that it is simple to describe protocols that cannot be enforced locally [93].

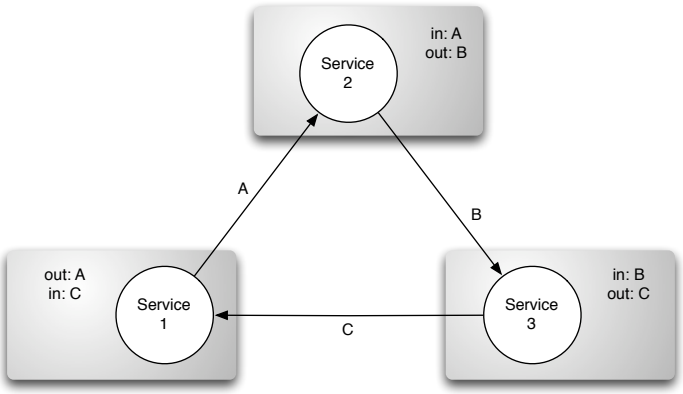


Figure 2.11: Service Centric Interaction Protocol

Both the complexity of the protocol and the method used to describe it can vary dramatically. On one hand the protocol description may only be implicit and based on the details of the implemen-

[4] we use the terms interaction protocol, coordination protocol and abstract process interchangeably

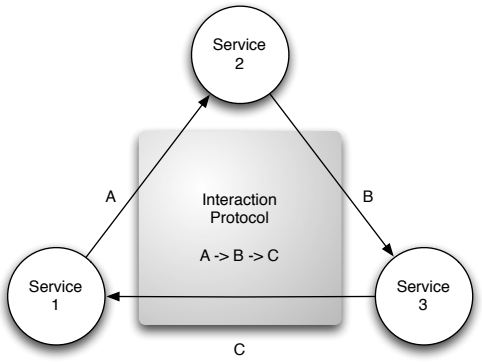


Figure 2.12: Global Interaction Protocol

tation of the service. Conversely the protocol may be explicit and take the form of anything from informal descriptions (eg. verbal, business documents) to formal, machine processable representations (eg. abstract BPEL [94], WS-CDL [21] or SSDL[95]). Whether implicit or explicit, all services have an interaction protocol that must be followed to communicate with them. Because interactions between services cannot normally happen independently of each other, explicit protocols have the benefit that they can be used by other services for deriving the correct interaction behaviour.

Benatallah et al. claim that protocol descriptions can have positive effects on service development, binding and execution thus simplifying the service life-cycle [96]. Protocols can be used to determine whether services are compatible with each other, replaceable or even equivalent [97, 96]. The latter question has been researched in depth by the semantic web community for some years and it demonstrates the difficulty of the problem the fact there have been no significant breakthroughs reported [98]. One area discussed in [96] is the advantage that 'protocol-aware' middleware can bring:

Protocol Monitoring If protocol descriptions are present, middleware can establish whether the sequence of messages exchanged adheres to the protocol. If violations are found the middleware can take reactive measures such as dropping the message or sending a fault message. This removes the burden from the application developer of having to deal with such issues.

Conversation Based Message dispatching Protocols can be used to control the logic for dispatching messages to application code. This accounts for the fact that the semantics of a message might change during a protocol and the fact that not every message need initiate business logic. Again this simplifies the application code making it easier to write and maintain.

Code generation A protocol can be used to develop more sophisticated stubs that 'know' how to interact with the service

Analysis Services that advertise protocol descriptions are amenable to a range of analysis methods. For instance, design time verification of compatibility and assessing the impact of service evolution. Service compatibility is an area addressed by this thesis and is discussed further in Section 2.4.

Research has been undertaken to try and identify a suite of Service Interaction Patterns [99]. In a similar vein to the way that the work on workflow patterns identified a number of scenarios that are often found within the structure of workflows [100], Service Interaction Patterns show common structure of the ways services interact. The work covers simple message exchanges and complex ones where, for instance, messages are relayed to a third party who interacts with the originator directly. The authors divide the set of patterns according to a range of criteria: the number of parties involved in the exchange which might be two (bi-lateral exchanges) or unbounded (multi-lateral exchanges); the maximum number of messages exchanged between the two parties which might be two (single-transmission interaction) or unbounded (multi-transmission interaction); and for two-party interactions whether the receiver of the request is the sender of the response (round-trip interactions) or (routed interactions). In Chapter 4 we will compare SSDL with these patterns and show those that we are able to express and those that we are not. There is a slight mismatch between the scope of Service 'interaction patterns' and Service 'Description Language' in that the former encapsulates more data than the latter. In particular, service interaction patterns are able to take time into account, a notion that SSDL is unaware of.

2.2.7 WS-CDL

Web Services Choreography Description Language (WS-CDL) allows the definition of multi-party contracts which describe the externally observable behaviour of Web Services and their clients (usually other Web Services), by describing the message exchanges between them[5]. Although not explicitly stated, WS-CDL defines such interactions from the global, party neutral perspective. The language is rich and makes use of imperative programming language constructs such as conditionals, repetition and grouping statements into blocks. The statements that are controlled by the language are interactions between parties in the choreography, such as transmitting a message. Sending and receiving messages is done between parties, over explicitly named channels and the data sent/received is stored in variables which are accessible to other interactions in the choreography. The state of a choreography at any one time can be considered to be the contents of these variables. If a choreography were considered executable these variables would need to be synchronised across partners, a problem identified in [93].

WS-CDL is based on π-calculus and there is an ongoing effort to formalise it by members of

[5]Taken from the WS-CDL Working Group Charter

the Working Group [101, 102]. The output of this work will be a formal model but it is not clear whether there will be tool support in addition to verify properties of a choreography. Others have attempted to create a formal model but have made some simplifying and unrealistic assumptions, namely that globally synchronised variables exist [103] or they are using a subset of WS-CDL [104].

One problem with global models of interaction patterns is that it is trivial to design an interaction that is not enforceable from a local perspective. That is, there are ordering constraints placed on sets of messages that are impossible to enforce without the additional coordination messages. For instance, the choreography "A sends a message to B and then C sends a message to A" is not enforceable as is. There is no way for C to know that A has sent the initial message. There are two methods found in the literature for dealing with such situations: firstly to ignore any constraint that is not locally enforceable [105], and secondly to detect such semantic errors and alert the choreography designer. An algorithm for detecting violations of local enforceability is presented in [93] and is applied to another language for specifying choreographies "Let's Dance" [106, 107]. There is no reason why the algorithm shouldn't be applied to choreographies specified in WS-CDL. "Let's Dance" itself is an interesting language; it aims to be at a higher level than WS-CDL (and BPEL) so avoids such procedural programming constructs such as variable assignment and instruction sequencing. The information that is captured within a Let's Dance model can be used to generate BPEL templates which capture the structure of the interaction but require some more information to be provided in order to be executable.

It is possible to bridge the gap between global service choreographies and local behavioural interfaces to services. Mendling and Hafner show how this is possible by generating executable BPEL for each partner in a WS-CDL choreography using a process known as endpoint projection [105]. However, when combined, the resulting BPEL processes may not adhere to the exact choreography for the reasons mentioned above: non-enforceable local constraints are ignored.

2.3 Workflow Management

Workflow is the computerised facilitation or automation of a business process. A business process is a set of one or more linked procedures or activities which collectively realise a business objective [108]. Business processes are valuable intellectual property (IP) for an organisation and much money is spent defining, documenting, refining and enacting them. The responsibility of enacting business processes is typically handled by a workflow management system (WFMS) or workflow engine for short. Many workflow engines are available for enacting intra-organisational business processes where factors such as ownership, network latency and availability are not an issue. Inter-organisational business processes must address these issues and fewer workflow systems exist in this domain[109]. In addition, as mentioned earlier, workflow engines targeted at Virtual Organisations require flexibility

as many different organisational structures and domains of control exist. This section will discuss the technologies and implementations that are relevant and will give an overview of a number of workflow engines in this arena.

2.3.1 Workflow Reference Model

The Workflow Management Coalition provide a Reference Model that gives an overview of a generic workflow system, identifying the core components and interfaces between them [110]. Most modern workflow management systems echo this design when viewed on a high level and the model has had influences on our design presented in Chapter 5. The model, shown in Figure 2.13, shows the main components and how they interact with each other through well defined interfaces numbered from 1 to 5. The definition of workflows is usually created using Process Definition Tools which may offer graphical editing of graph like structures to depict the workflow. These are then saved into a Process Definition Language and loaded into the enactment service through Interface 1. The Process Definition Language is executed/interpreted by the Workflow Engine. In fact there may be multiple Workflow Engines which coordinate to execute the workflow. This coordination takes place through Interface 4. Interface 5 allows monitoring and administration of workflows that are currently executing. Interfaces 2 and 3 are used during the execution of the workflow to actually 'do the work'. Interface 2 provides mechanisms for human engagement in the workflow whereas Interface 3 allows the invocation of other applications (which may or may not be local to the workflow engine). Ideas for distribution similar to those encompassed in Interface 4 have been presented elsewhere to allow coordination between heterogeneous workflow engines [111].

The WfMC are still active and produce a yearly report on the state of workflow and business process modelling. They maintain a list of products that are known to support the different interfaces defined in the reference model [112] and have recently standardised a language for serialising workflows, XPDL [113].

2.3.2 BPMN

Business Process Modelling Notation [114] is high level modelling notation for capturing business processes developed by the Business Process Modelling Initiative[6]. The aim is to provide standardised notations that can be understood by all parties involved with business process design, execution and monitoring. The notation consists of diagrams which describe tasks, processes, decisions, data and control flow and artifacts. There are also standard methods for describing sequences of interactions between different aspects of the system using 'swim lane' diagrams. Such swim lane diagrams can be also be used to show how a number of discrete systems interact with each other. In this

[6]The BPMI merged with the Object Management Group in 2005

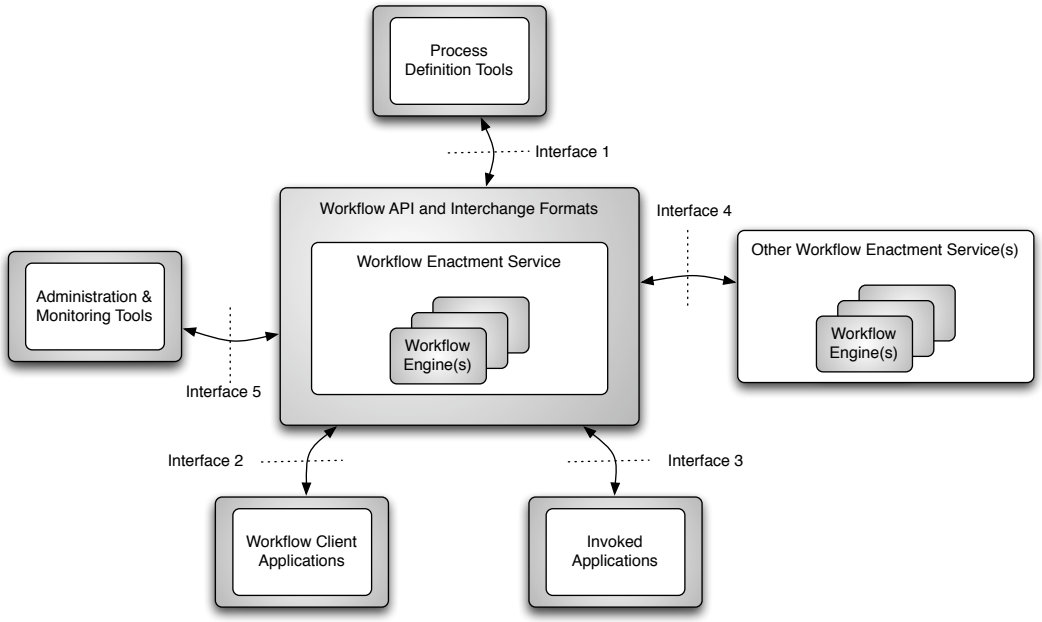


Figure 2.13: WFMC Reference Model. Reproduced from [110]

case, each system has its own 'lane' within the 'pool'. As such, swim lane diagrams are, informally similar to UML Sequence Diagrams but they allow a richer set of control flow within the diagram.

Owing to its high level nature, BPMN diagrams are not directly executable. Instead, it is possible to map them onto other executable languages such as BPEL[115]. The abstract nature of the language which provides its power, also provides its weaknesses. It is possible to model the same interaction in a number of ways and it is not clear that the same executable code would be generated each time. This may introduce subtle behaviours which were not intended in the model. It is not possible to perform verification on the model (although it might be possible on one of the generated executables). Despite these and other problems, BPMN appears to be gaining traction as the modelling notation of choice and is achieving significant vendor support [116, 117].

2.3.3 BPEL4WS

The aim of Business Process Execution Language for Web Services (BPEL4WS, usually shortened to BPEL) [94] is to provide a standard for specifying business process behaviour and business process interactions, for applications composed from Web services. We will give an overview of BPEL here and Section 3.3 will compare and contrast it to the Task Model which we will present in the next Chapter. There are two types of a BPEL process, executable processes and abstract processes. The aim of an executable process is to fully describe the workflow, including all relationships between activities, data flow and exception handling. The resulting process can be executing by a runtime

environment and given input will interact with other services and (hopefully) complete. An Abstract BPEL process, however, is not directly executable. It describes the behaviour of a service from the perspective of an external observer but does not necessarily contain all of the information that an executable process contains. An abstract BPEL process is similar to the SSDL description of a service that we will present in Chapter 4.

A BPEL process is constructed out of a number of elements that control the order of execution: sequence, while, switch, flow, pick. The first three items are self explanatory whereas the last two are a little less obvious. Flow provides support for arbitrary directed graph based structures. Performing splitting and joining is possible with a flow construct and it is possible to specify intricate links and conditions between activities within a flow to control their execution. The pick construct waits for one of several events to occur and then executes the activities associated with that event. Once one of the events has fired in a pick construct the others are deactivated. Within the constructs listed above, it is possible to control the execution of a number of types of activity: invoke, receive and reply. Invoke calls a partners Web Service, either synchronously or asynchronously waiting for a reply if it is the former. Receive models the other end of an Invoke activity and consumes a message, possibly creating a new instance of the BPEL process to deal with the message. The reply activity sends back a previously received receive activity.

There are two important concepts that relate to the activities described above: variables and partner links. Within a BPEL process, variables are used to store the current state of the execution. Each activity can produce and consume variables, e.g. the input and output of an invoke activity are variables. Variables are XML snippets and can have structure which can be queried using XPATH[118] and can either have global or restricted scope. It is possible to copy portions of a variable from one to another using the copy construct. Partner Links define the relationships between this process and other partners in the interaction. A partner link is defined between two WSDL PortTypes with one PortType being implemented by the BPEL process and the other by the partner. The partner link defines which one is implemented by whom and gives the link a name that can be referred to by activities. Each activity will refer to this partner link and it is used to uniquely identify the operation that is being invoked (partner link type, port type and operation is unique).

Message correlation in BPEL is handled by either WS-Addressing headers or by BPEL correlation sets. Correlation sets are sets of properties that should uniquely identify an instance of the process from the contents of a message, order number for example. Using WS-Addressing is more straight forward as unique identifiers for the conversation will be generated. BPEL also provides rich fault tolerance support through the use of scopes, fault handlers and compensation handlers. Scopes are hierarchical; a scope can have multiple subscopes, each with its own structure. Each scope can define compensation handlers which are intended to undo the work that had previously been done

in the scope if an undesirable situation is reached. A compensation handler can be invoked from a subsequent scope and is equivalent to the rollback of a transaction processing system. Fault handlers are local to a scope and are used for dealing with in-flight faults and are similar to exception handlers of programming systems.

Many enactment engines for BPEL exist but of particular interest to us are the ones described by Baresi [119] and Chafle et.al. [120]. Both of these describe decentralised, or distributed, enactment engines for BPEL. There are significant challenges associated with producing a decentralised enactment engine for BPEL due to the rich set of constructs in the language and the presence of global variables. To produce a generic decentralised engine the global variables must be synchronised which is not an easy problem to solve. The former work avoids this problem by explicitly stating that the workflow must not contain any global variables. Whilst it is possible to write BPEL workflows that do not have any global variables it is not clear how this effects the expressiveness of the language (we will discuss the expressiveness in the following Chapter). The latter work will only distribute the enactment if it is written as a tree structure of BPEL processes. That has the effect that each branch of the tree is independent and can be controlled by a different engine. Whilst this does not place any restrictions on the constructs used it does place significant restrictions on the structure of the workflow.

2.3.4 JOpera

JOpera is an autonomous process execution platform for Web Service compositions [121][7]. The platform consists of a design time editor for workflows, a runtime environment that supports distributed execution and monitoring tools to control the execution. The runtime environment consists of a number of distinct components which can be deployed on different nodes within a distributed system. The *Navigator* receives requests from the process queue to instantiate a new process and schedules the invocation of tasks depending on the state of the execution. When a task is deemed executable a service invocation request is placed in the task queue and is retrieved by a *Dispatcher*. The Dispatcher is responsible for interacting with the service provider through the appropriate mechanism (eg. by exchanging SOAP messages or performing a Java method invocation). When the dispatcher has completed its execution it notifies the Navigator through an event queue allowing the Navigator to schedule the next tasks. This decoupling is natural as the different components have varying granularities and the invocation of the remote service may last significantly longer than the time taken by the Navigator to schedule it. The decoupling also enables the platform to scale to handle both more Dispatchers and Navigators. This scaling is able to happen autonomically through the use of control algorithms which are tuned with QoS requirements [122]. The architecture of JOpera is very similar to that of DECS in the use of queues to connect components responsible for scheduling

[7]For the purposes of this thesis the terms process execution and workflow enactment are treated as the same.

with those responsible for message dispatching. The main difference between the JOpera approach and our own is in the target network configuration: JOpera is intended for clusters of machines with high bandwidth, low latency interconnections. DECS on the other hand is designed to execute across organisational domains. The result is that whereas a distributed DECS system replicates the whole workflow engine, JOpera replicate the components of a single workflow engine. Other differences are that the distributed nature of a workflow within DECS is explicit but the philosophy behind JOpera is for implicit distribution giving the illusion of a single instance.

2.3.5 SELF-SERV

SELF-SERV offers a platform for declaratively composing Web Services into workflows using UML Statecharts [123, 124, 125]. The execution platform for SELF-SERV offers two interesting properties: the use of service communities and peer-to-peer enactment facilities. Service communities are a method of offering alternative services when multiple equivalent ones exist. Each service in the community has to provide a mapping from the syntax of its interface to that of the community and then at runtime the platform selects a suitable service according to QoS parameters. The QoS parameters may change with each execution and take into account service metrics or parameters of the request.

Peer-to-peer enactment in SELF-SERV removes the need for a centralised coordinator and scheduler. At deployment time the workflow is analysed and a distributed execution schedule is created. This includes routing tables and event triggers to schedule service invocation. In essence, following each service invocation the SELF-SERV platform sends the data generated to any downstream services that require it to execute. The data is sent directly rather than through centralised coordinator. Event notifications are sent this way too, informing downstream services that an upstream service is now in a particular state. Each service is invoked when a certain set of states is reached and the data is available. As this decision can be taken locally for each service the system has been shown to scale well [123]. The amount of data that must be transferred has also been shown to be reduced by using such peer-to-peer enactment.

One problem with the approach taken by SELF-SERV is that every service to be used within a workflow must be wrapped with a *coordinator* that is able to receive the event notification and make the decision on when the service should be invoked. The application provided for this is only able to work on Web Services written in Java. This means that every service must be implemented in Java and the source code and execution environment must be available to the workflow designer. This precludes the use of services provided by third parties as it is unlikely that access will be provided to the source code or execution platform. In fact it negates some of the advantages of Web Services, namely that they are platform and location independent. The approach taken in DECS is similar to that in SELF-SERV, except that DECS is able to invoke services that reside in a different

organisational domain. Effectively, if one instance of DECS were deployed per service (at the same location as the service), there would be little difference between the two approaches.

2.3.6 Other Related Developments

DecSerFlow [126] is a declarative workflow language aimed at removing the problems that arise with most procedural workflow languages, for instance having to specify all possible interleavings of activities. DecSerFlow describes a set of activities along with conditions that must be either fulfilled or avoided. The workflow will complete when all the conditions are met. This is an interesting approach to workflow specification and uses a graphical notation for designing workflows and has a grounding in Linear Temporal Logic (LTL) [127]. The graphical nature overcomes the problems associated with LTL systems: they are only understandable by experts whilst still allowing the verification of properties. In [126] it is described how to generate Buchi automata [128] from a DecSerFlow workflow and this is used as the basis for executing the workflow. However, it is not clear whether the execution is restricted to a single machine or could be distributed across multiple machines. One of the problems to overcome is that all the constraints must be considered at any moment of service execution. It will be non-trivial to synchronise the states across multiple machines that are performing the enactment.

CrossFlow [129] is a workflow management system that invokes tasks using Java RMI rather than Web Services. It was developed prior to the Web Services standards being popular and achieves distribution using the WFMC's Interface 4 for inter-workflow engine communication. This allows one workflow engine to delegate work to another at runtime. CrossFlow attempts dynamic matchmaking of tasks at runtime using service contracts to define the capabilities of service providers.

Taverna is a workflow engine developed principally for the bio-informatics field [130]. It has a large user base, an intuitive user interface and is able to interact with services that support SOAP, Java RMI, Beanshell and other technologies. However, it is targeted at scientists who run it on their personal workstation rather than on a remote server. Hence it does not offer distributed workflow even though this might vastly reduce the network traffic needed to execute a workflow: running part of the analysis on the local machine and part on a node close to the large data sets that bioinformaticions use might result in significant efficiency gains.

To summarise, there are two common ways of achieving distributed workflow enactment with the difference being where the coordination logic is hosted. In the first category the coordination logic is co-located with the Web Service being coordinated. This method, implemented in SELF-SERV and BondFlow [131] has the advantage that it is quite lightweight and efficient. The main drawback is that it requires access to the Web Services being coordinated and needs platform dependant code for each service in the workflow. It is also not clear how adaptable this approach is as it requires design time definition of the routing of messages between services. The second approach, taken by

jOpera, CrossFlow and ourselves is to provide coordination logic that is not co-located with the services being utilised. This has the advantage of being more flexible and not being dependant on the implementation platform of each service. However, it will not necessarily be as efficient as the former approach in terms of the number of messages required to coordinate an execution.

2.4 Analysis of Workflows and Protocols

Analysis aims to determine whether the subject satisfies some condition. When considering workflows and protocols we are mainly concerned with *safety* and *liveness* conditions [132]. Informally, a safety property stipulates that "bad things" do not happen and a liveness property stipulates the "a good thing" will happen (eventually). The question is how these properties relate to workflows and interaction protocols.

Within the domain of workflows safety conditions typically relate to deadlock (It could be argued that the showing the absence of deadlocks is a liveness issue. However as we are interested in showing both safety and liveness the distinction is somewhat arbitrary). A deadlock may occur when a task is dependent on two mutually exclusive events occurring or when circular dependencies exist. That is, when Task A is dependent on Task B completing and Task B is dependent on Task A completing. Such circles may be arbitrarily long and may have conditional behaviour too making them difficult to diagnose by inspection alone. Within workflows, liveness typically means that the workflow will complete executing eventually, no matter what execution path the workflow takes. In the literature this is sometimes referred to as "dead path detection". Unbounded recursion and infinite loops are other examples of violations of liveness properties.

When considering protocols between different Web Services, regardless of their implementation, the safety and liveness conditions are sometimes referred to as *compatibility*: whether one service is compatible with the other. Lack of compatibility may be because one service emits a message that the other one cannot process or that one service requires a message that the other never sends. The former can be considered violation of a safety property and the latter relates to liveness.

Decker and Weske introduce a more complete notion of compatibility in [133]. They define *structural compatibility* as the ability to receive every message that the other service is able to send. When the message sets are complete, as in the latter service is able to send every message the former service is able to receive the relationship is referred to as *strong structural compatibility*. This is a powerful relationship but does not take into account that a service may offer functionality that the other service may choose not to use. If every message that can be sent can be received this is referred to as *weak structural compatibility*. Although weak structural compatibility is most commonly required for systems to communicate, the authors also define *minimum structural compatibility* if there is at least one message sent that can be received. This is likely to be useful for describing systems where

dropping unrecognised messages is acceptable.

Following on from structural compatibility which only considers the type of the message and ignores protocol state, Decker and Weske discuss *behavioural compatibility* which takes into account the temporal requirements between messages within a protocol. Martens analyses the observational behaviour of workflows written in BPEL by converting them into petri-nets [134]. A Petri net is a directed, connected bipartite graph where each node is either a place or a transition. Tokens occupy places and when there is at least one token in every place connected to a transition, the transition is enabled. An enabled transition may fire, removing one token from each input place and depositing one token in each output place. Due a combination of their graphical, graph based nature and and their formal definition, Petri nets have been extensively used to model business processes. In Martens' work, the workflows are composed together and corresponding input/output places in the petri-net are merged into a single transition. The global process that is created is shown to be free of deadlocks if the final marking is always reachable. One disadvantage of this approach is that it requires strong structural compatibility to work. Canal et al. use a method of analysing protocols which is similar to ours, namely the formalisation of the protocol using π-calculus[135]. The main advantage of using π-calculus is the availability of link passing to model mobility. This means that the communication channels between interacting processes do not need to be statically defined but can be established at runtime, akin to dynamic binding. The compatibility notion by Canal et al. requires that processes complete, i.e. there are no sending receiving actions left to be performed. The disadvantage of their approach is that it requires strong structural compatibility and it is only defined for bi-lateral settings (two processes).

Soffer and Ghattas analyse inter-organisational business processes in [136] using the notion states that are shared between two workflows. They provide a Generic Process Model which is able to define workflows and when combined the shared states between two workflows can be identified. As long as the temporal dependencies between the shared states are the same in both workflows they show that the workflows are compatible. However, they only describe the process for two parties and it is not clear how this approach will scale. The dependencies between shared states when there are three or more parties is likely to grow at a greater than linear rate making analysis difficult.

Benetallah et. al. show that it is possible to abstract conversations into different types of actions [137]. They build on Finite State Machines and identify completion abstractions (moving from one state to another), compensation and resource locking operations and activation abstractions (describing a transition's triggering features) as useful abstractions for modelling protocols. Preconditions including temporal functions are used within the activation abstractions to describe when a transition is enabled. They also present an architecture for managing conversations and hint at methods of generating the conversation models and analysing them.

The act of producing a formal model of BPEL has been a hot research topic for a number of

years [138]. Many different approaches have been taken, using different formalisms and aiming to cover a different amount of the BPEL specification. Three of the common approaches are to use Petri nets, process algebras or the SPIN model checker. The advantage of mapping BPEL to a Petri net is that not only has a formal model been constructed, but the verification techniques and tools developed for Petri nets can be exploited in the context of the BPEL process. A number of authors such as Schmidt and Stahl [139] and Martens [140, 141, 134] use Petri nets to reason about BPEL. The former work presents examples and a transformation from BPEL to Petri nets and the same authors have constructed a tool to automate the process [142] and verification of properties has been shown using the LoLA tool [143]. Verbeek and van der Aalst focus on the structured activities of BPEL and present a mapping from these to class of Petri nets called workflow nets [144]. Ouyang et. al. go further and provide a formal semantics for BPEL by providing a mapping from all control flow constructs of BPEL to Petri nets [145]. They also discuss tools that are able to automate the process and detect undesirable situations such as unreachable activities.

SPIN [146] is a tool to verify programs written in Process Meta Language (Promela) and properties specified in Linear Temporal Logic (LTL). Provided that the Promela program is bounded, SPIN can check if the program satisfies the LTL property. A number of people have developed transitions from (part of BPEL) to Promela, for example Fu, Bultan and Su [147] who first translate the BPEL process into a guarded automaton and subsequently into BPEL. They are also able to deal with XML data and XPATH expressions by expressing these in Model Schema Language (MSL) which has a mapping onto Promela. Nakajima has presented similar work in [148] and also applied his Promela models to detect information leakage from BPEL processes.

Kramer and Magee have developed a process algebra named FSP (Finite State Process) where each FSP represents a labelled transition system [149]. In collaboration with Foster, they have also developed a tool called Labelled Transition System Analyser (LTSA) and provide an extension to LTSA for analysing BPEL [150]. The tool translates BPEL activities into FSPs and then uses the existing tool to perform the analysis and checks for safety and liveness properties as well as properties expressed in LTL. As FSP has a formal semantics this work can be said to have produced a formal model for part of BPEL. In [151, 152] Foster et. al. show that it is possible to combine the BPEL with either Message Sequence Charts (MSC) or WS-CDL and use LTSA to analyse the resulting system. The MSCs or WS-CDL are translated into FSPs and again LTSA is used to determine whether the BPEL process is consistent with the MSCs or WS-CDL representation respectively. Foster et. al also note [153] that in some cases model checking is not enough. Although model checking is able to determine some deficiencies in BPEL processes, others only become apparent at runtime. Such problems are a result of the choice of synchronisation primitives, the threading model of the server etc. They proceed to show a model that takes into account execution resource constraints to more accurately depict the process.

2.5 Formal Notations and pi-calculus

The π-calculus [154] is an algebra for describing and analysing the behaviour concurrent systems. A π-calculus system is described in terms of processes, channels and names. Processes are independent of each other and communicate using channels which connect them. Each channel is referred to by a name and the communication unit along a channel is a name. A name is the most primitive unit of addressing in π-calculus. Processes are built from the following action terms (also known as prefixes) and operators:

- Send [$\bar{x}<a>.P$] - Send the name a along channel named x and then execute process P. Sending an empty message over channel x is shown as $\bar{x}.P$
- Receive [$x(b).Q$] - Receive name b down the channel named x and then execute Q. This has the effect of binding all occurrences of x in process Q. Receiving an empty message on channel x is shown as $x.P$
- Choice [$P1 + P2$] - Execute exactly one of the processes P1 and P2. The execution of one half of this expression precludes the other half from ever being executed. This operator is associative and commutative and the choice is non-deterministic.
- Parallel Composition [$P1 \mid P2$] - Execute the processes P1 and P2 in parallel. These two processes may communicate with each other via named channels. This operator is associative and commutative.
- Sequence [$P1 . P2$] - Execute Process P1. When it completes execute process P2.
- Replication [$!P$] - Execute an infinite number of copies of P in parallel. It is possible to use replication to simulate recursion and therefore not necessary to include a separate operator.
- Restriction [$(\nu x)P$] restrict the scope of x to the process P.

There are two special actions that exist in the π-calculus which should be considered: τ and 0. Firstly, the τ action denotes an internal unobservable action. This action may perform transformations of data or other such actions which are not externally visible. Secondly, the 0 operator signifies explicit termination, for instance, $P.Q.0$ means execute process P, when it completes, execute process Q and then stop. The 0 is often omitted for brevity, simply writing $P.Q$ but where it adds clarity or cannot be implied from the context it is included.

Two forms of π-calculus exist: monadic and polyadic. In the monadic form of π-calculus, only one name may be sent along a channel in an execution step. For instance, $\bar{x}.<a>.P$ is allowed but $\bar{x}.<ab>.P$ is not, assuming that a and b are separate names. The polyadic form of π-calculus allows multiple names to be sent and received along a channel in one computation step. It can be shown

that the polyadic form is necessary and that the natural monadic abbreviation $\bar{x}.<a>.\bar{x}<b>.P$ is not equivalent to the polyadic term $\bar{x}.<ab>.P$ [155]. This is because more than one process might be receiving names on channel x. A common work around is to first transmit a fresh channel and then send a and b along that channel: $\bar{x}.<y>.\bar{y}<a>.\bar{y}<b>.P$. This ensures that both names will be received by the same process.

Computation in π-calculus is defined by a set of reaction rules which describe how a system P can be transformed into P' in one computational step ($P \rightarrow P'$). Every computation step in the π-calculus consists of communication between two terms (which may be part of separate processes or the same process). Communication may only occur between two terms which are unguarded (that is they are not part of a sequence prefixed by an action yet to occur) and not alternatives to each other. Consider $P = (...+x(b).Q) \mid (...+\bar{x}<a>.R)$, when the system is in its initial state P, two parallel processes are executing, and the latter sends the name a along the channel x. The former process receives a along channel x as the sending and receiving terms are complementary and unguarded (said to form a redex). The action of receiving a has the effect of substituting a for b in the process Q and the transformation $P \rightarrow P'$ has occurred where $P' = \{a/b\}Q \mid R$. The substitution is denoted by $\{a/b\}$ in the process P'. A side effect of this communication occurring is that the alternatives (denoted by ...) have been discarded and any communication that they would have performed has been pre-empted. We have now performed one computation step in the system and the system is in a new state.

In many cases there may be multiple states which a process can be transformed into. For example, following process $P = (\bar{x}<a>.Q) \mid (x(b).R) \mid (x(c).S)$ there are two transformations possible $P \rightarrow P'$or $P \rightarrow P''$. In the process P, name a is being sent along the channel x but can only be received by one of the other two parallel compositions. Therefore after state P, the following states are $P' = Q \mid \{a/b\}R \mid (x(c).S)$ which assumes that the name a is received by the middle composition causing a substitution of a for b in process R; or $P'' = Q \mid (x(b).R) \mid \{a/c\}S$ where a has been received by the other composition and is substituted for c in process S.

In order to perform conditional branching we can test a name to see whether it holds a certain value. For instance, $[x{=}accept]response(x).P + [x{=}reject]response(x).Q$ performs process P if the name received on channel response is 'accept' or process Q if the name received on channel response is 'reject'. We use a few notational conventions in the thesis: to denote the parallel and serial execution of the prefixes $\pi_i, i \in I$ we use $\Pi_{i \in I} \pi_i$ and $\{\pi_i\}_{i \in I}$ respectively. To denote the non-deterministic choice of prefixes or $\pi_i, i \in I$ we use $\sum_{i \in I} \pi_i$.

A central notion in the π-calculus is equivalence: whether two processes exhibit the same behaviour [156]. The description that follows is an informal introduction to bi-simulation in the π-calculus as a full formal discussion is outside the scope of this thesis. Let us consider two processes,

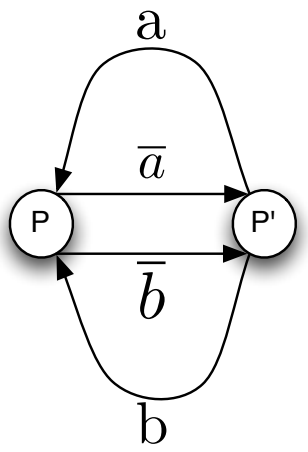


Figure 2.14: Process P

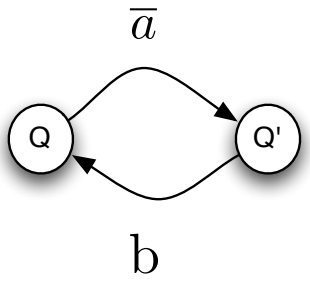


Figure 2.15: Process Q

P and Q. Finite State Machines for the two processes are shown in Figures 2.14 and 2.15 respectively. Both are processes that send and receive over two channels named a and b. The π-calculus representations of the processes are shown below:

$$
\begin{aligned}
P &= \overline{a}.P' + \overline{b}.P' \\
P' &= a.P + b.P \\
Q &= \overline{a}.Q' \\
Q' &= b.Q
\end{aligned}
$$

Process P can either send a message over channel a or over channel b and then move into state P'. From this state it can either receive a message over channel a or channel b causing it to move back into its original state. Process Q can send a message on channel a and then receive a message on channel b.

We say that one process simulates another when the same transitions can occur in the source as in the target. That is, if $P \mathcal{S} Q$ then every transition in Q must be present in P. In this case it is true that $P \mathcal{S} Q$. The converse cannot be said to be true. Not all transitions in P are possible in Q, therefore $Q \not\mathcal{S} P$. When the reverse is true so that $P \mathcal{S} Q$ and $Q \mathcal{S} P$ we say that the two processes are bi-similar, $P \sim Q$.

In fact there are different measures of 'equivalence' for example, strong and weak equivalence. The description given above is actually for a 'strong bi-simulation' which is a very fine equivalence system - not many processes end up being equivalent. In addition, internal actions (τ) are not handled as you would expect. Intuitively, two processes that only perform internal actions should

be considered equivalent: from an observer's perspective they cannot be distinguished. $\tau.\tau.0$ and $\tau.0$ both only perform internal, unobservable actions, however, according to strong bi-simulation they are not equivalent.

In order to capture the fact that internal actions should be ignored we can use a notion of equivalence called 'weak equivalence'. If P and Q are weakly bi-similar then a τ transition by P can be matched by zero or more transitions by Q. An observable transition in P can be matched by one or more transitions in Q as long as one of them is equivalent to that transition in P. i.e. if α is a transition that causes $P \xrightarrow{\alpha} P'$ then there must be a transition in Q such that $Q \rightarrow ... \xrightarrow{\alpha} ... \rightarrow Q'$. Weak bi-similarity is a coarser equivalence relation than strong bi-similarity: a strong bi-simulation is also a weak one but the converse does not hold.

In addition to strong and weak equivalence there are other factors which influence bi-simulation. Early and late bi-simulation are introduced by Milner, Parrow and Walker in [157]. The difference between the two forms is concerned with the time when name instantiation occurs compared to the choice of which derivative of a process to execute. In late bi-simulation the choice of derivative of a process occurs before the choice of the value with which to bind a name x. The reverse is true for early bi-simulation. One problem with early and late bi-simulation is that the equivalence relations are not preserved by all operators: notably the input prefix can cause failure. For instance, according to early and late bi-simulation $[x = b]\overline{b} < b >$ and 0 are equivalent as neither can perform any action. However, if we add the input prefix $a(b)$ to the front of each process then they are no longer equivalent. This is because the input prefix has instantiated the name x with the value b causing the matching construct to evaluate to true. To overcome this problem, Sangiorgi introduced open bi-simulation in [158] and it is this form that we will deal with in this thesis.

In [159] the authors present two languages based on process calculi one to model choreography, the other to model orchestration. They demonstrate how a given orchestration can be shown to be conformant to the choreography using a bi-simulation based approach. They claim that their languages are inspired by WS-BPEL and WS-CDL but do not demonstrate any close links to these languages. A direct mapping from (Abstract) WS-BPEL and WS-CDL might be possible but it is not presented in the work.

2.6 Java Enterprise Edition

This Section will give a brief background on Java Enterprise Edition which we will use in Chapter 5 as our implementation platform. The intention is to introduce the reader to some of the concepts and terminology so that the design decisions discussed in Chapter 5 will be easier to understand.

Sun Java Enterprise Edition (Java EE) is a component architecture, suite of runtime services and set of standard programming interfaces and abstractions [160]. The aim is to help the developer

write and deploy server side applications. The architecture of a Java EE application is similar to a standard three tier application but the business logic tier is divided into a number of separate pieces: entities, business processing logic and asynchronous processing logic. The Java EE standard also includes technologies to handle the presentation layer but those do not concern us here. Instead we will focus on the technologies used within the business logic tier (Enterprise Java Beans, EJBs) and some of the runtime services that offer support.

Let us work from the 'bottom up' of a three tier application and start with the database. Java EE uses a relational database to persist the data within the application. This database could be a lightweight 'in memory' database or could be a large, stand-alone database running on a different machine. A specification known as the Java Persistence Architecture (JPA) defines how the application will interact with the database. This specification provides a method of persisting and retrieving 'plain old Java objects' (sometimes called POJOs) to the database. These POJOs are *Entity Beans* and are like any other Java class except they have JPA metadata instructing it how to persist the object to the database. For instance, which member variable within the Java class maps onto which database column; which database table the object should be stored in; whether the Java class is stored in a single table or a number of tables joined together. Developers can use these objects within their application in exactly the same way as any other Java object. The only difference is that it is possible to store the object when desired using the functionality provided by the JPA.

The business logic of the application is provided by a *Session Beans*. Again, in Java EE, Session Beans are plain old Java objects but they may have metadata associated with them to provide access from outside the container, namely a Remote Interface. This interface can be looked up in the directory service included in a Java EE implementation, JNDI. It is also possible to provide a Local Interface which allows other Session Beans to make use of the Bean within the application. It is Session Beans that define the business logic of the application. They expose business methods to the tier above and clients outside the application. A Session Bean will make use of Entity Beans during request processing to look up information from the database and persist any new data that must be saved. Two types of Session Bean exist: stateful and stateless. Stateful Session Beans (SFSB) maintain the conversational state when used by a client. That is, every invocation from a particular client will be routed to the *same instance* of the Session Bean. Any member variables that are set within the Stateful Session Bean will be present when the next method is invoked. This conversational state is not persisted to the database but held in memory and will be lost when either the client disconnects or the EJB container crashes. Stateless Session Beans do not maintain any conversational state and all the information necessary to process the request must be present in the request itself. Any member variables set on previous requests will not be present on the following request. Stateless Session Beans are considered more lightweight than Stateful ones as a few Beans

can process many client requests. However, the amount of data passed will increased as it must include all the information required to process the request and in practise the state must be stored somewhere and recreated each time. Different applications have different requirements and will find the use of Stateful or Stateless Session Beans more appropriate at different points.

Message Driven Beans (MDBs) differ from Entity and Session Beans as they provide an asynchronous way of invoking functionality within an application. They are often used to provide integration points for legacy applications who can send messages to Message Driven Beans via the Java Message Service (JMS). A MDB 'listens' to a JMS queue or topic and the message is processed by the MDB as soon as a bean becomes available from the pool (MDBs, like other types of EJB are pooled within a container and the size of the pool can be altered to suit application requirements). Once the MDB has received the message, it processes it which may involve the use of Session Beans and Entity Beans and then it completes.

The three types of Enterprise Java Bean described above are only one aspect of the Java EE specification. It also includes other items such as transactional behaviour, security, management APIs and technologies that work at the presentation tier. Transaction management is a complex topic which we will not cover in detail. However, it is worth mentioning that Java EE provides declarative transaction management that can be highly configured to suit an applications requirements. It is also possible, if required, to explicitly manage the transaction using the Java Transaction Service (JTS).

The Java EE specification is just that, a specification. In order for it to be used there must be an implementation of it. In theory, an application is portable across any Java EE compliant application server with very little effort. In practise this is not always the case as many application servers will provide extra functionality that is not defined by the specification. A number of compliant application servers exist in both the Open Source and commercial arenas, such as JBoss[161], Sun Glassfish[162], IBM Websphere[163] and Oracle Weblogic[164].

2.7 Summary

Of particular relevance to this thesis is the work done by Lethbridge to describe different structures of Virtual Organisations [12]. They show that there is no single structure that can dictate success, rather that the structure will be dependent on the domain and operational requirements of the VO. This provides our requirement for flexibility in enactment of the workflows that form the execution of the VO, a requirement that BPEL does not meet. Baresi [119] has shown that it is possible to provide a decentralised enactment engine for BPEL but only if significant restrictions are placed on the BPEL constructs used, including not using global variables. We feel that these restrictions significantly limit the language and it is not clear how they effect the expressiveness. DecSerFlow

[126], a declarative language for describing workflows has certain similarities to our language in that it is possible to verify that undesirable, application specific, situations to not occur. For instance, the sending of goods after payment has failed. However, for a similar reason to BPEL it is not straightforward to enact it in a decentralised manner.

When considering protocol descriptions, WSDL [165] is the current standard for Web Services. However, WSDL is not able to describe the relative ordering of different operations that a Web Service exposes. In Chapter 4 we will present a language that is able to describe such ordering. In contrast to WS-CDL which can be used to describe the ordering from a global perspective, our approach provides a service centric viewpoint. Let's Dance [107] also provides a global perspective of the protocol description but is particularly relevant as it has a basis in π-calculus and a formalism similar to ours.

There are many enactment engines for workflows, most are centralised but some offer decentralised enactment. jOpera [121] has a similar architecture to ours but it is aimed at a cluster environment where high bandwidth, low latency network connections exist between each node. Based on the availability of such connections they divide their system into two components which can be replicated independently. The intention of a cluster of jOpera nodes is to appear as one single entity, which is different from our requirement of explicit distribution for different parties within a VO. SELF-SERV [123] provides such a distribution pattern and is based on peer-to-peer messaging principles. Their coordinator is very lightweight but can only coordinate a single service. The intention is to 'wrap' each service in a coordinator but this requires having access to the source code and deployment infrastructure of the service. Our approach is different and allows an engine to coordinate multiple services without needing access to the source code or deployment infrastructure.

Chapter 3

Workflow Modelling

3.1 Introduction

The Business Processes that an organisation own are valuable intellectual property (IP) for the organisation; they define in an unambiguous manner how the organisation responds to outside stimulus, operates and reports. It should be possible for someone completely unfamiliar with the organisation to study the business processes and learn the way the business functions on a day to day basis.[1]

Although business processes are valuable IP for an organisation, they are not directly executable. They capture what steps a business takes to achieve a goal in abstract terms that are often technology agnostic. There are many technologies available for representing business processes in an executable format but workflow is one of the most common. Business processes are usually defined graphically using a notation similar to flow charts [166, 167, 114] and given that workflow often has graphical components too it is possible to see the similarities. The graphical representation which will have certain, language specific, semantics is easier for non programmers to design, implement and monitor the workflow.

Workflows are usually executed by a piece of middleware [168] known as a workflow engine (sometimes the words process engine or enactment engine are used to mean the same thing). The workflow engine is responsible for managing the definition of the workflow, exposing it to other applications and managing each running instance of the workflow. Many workflow engines are similar to an interpreter for an interpreted programming language, that is they evaluate each statement (usually termed a task in workflow parlance) and execute those tasks that fulfil certain criteria. In some cases the workflow might be compiled to either executable code or an intermediary language such as bytecode [169]. Even in these cases there is usually something that is equivalent to a workflow engine to provide non-functional properties such as fault tolerance, security and provenance.

[1] It should be noted that not everything will be defined as a business process, for instance the process behind making some decisions may be too complex or impractical to define as a business process

As described in Chapter 2, Web Services provide a good basis for performing application integration in heterogeneous environments. Such an environment will be present in any single organisation, let alone a set of autonomous organisations. Any workflow system aimed at facilitating the management of Virtual Organisations should be able to interoperate with Web Services.

There is no 'one size fits all' solution to execution of workflows within VOs. Too much is dependant on the structure of the VO, whether there is a dominant party; where the services are hosted and what the security requirements are; the regulatory requirements of the industry. Clearly a VO that is configured as a Star Alliance will have different enactment requirements from one that is operating as a Parallel Alliance. Flexibility must be provided during the lifecycle of a workflow too. It is likely that the business processes will evolve over time and it must be possible to adapt the workflow in order to respond to these changes.

As workflows become more complex it is increasingly likely that errors will be introduced. Such errors might include unreachable tasks or deadlocks when the execution progresses along certain paths within the workflow. It is desirable to be able to analyse a workflow and ensure freedom of such situations.

It is possible to distill the descriptions above into set of requirements that a solution for enactment of workflows within a VO must address:

1. Allow flexibility in the enactment model
2. Be able to interoperate with heterogeneous applications
3. Be amenable to verification of safety and liveness properties

There are other requirements which are less specific to the domain of VOs but are generally considered good practice in software engineering. For instance, re-use of common elements of the workflow should be possible. This requires a modular approach and encourages the designer to consider the correct level of abstraction for each workflow.

A method of dealing with non-functional requirements should be included. In some cases, such as application level fault tolerance, these may be directly captured. It should be possible to capture other non-functional requirements, such as security and specify these in an independent manner. It should be possible to refactor the workflow at a later date, responding to changes in the business process or market conditions. Finally the language used to represent the workflow should be sufficiently expressive allowing a natural representation of the business process. Although it is easy to focus on the main requirements listed above, these general requirements should not be forgotten.

One of the contributions of this thesis is the presentation of the *task model* as a solution to the above requirements. This includes various representations of the task model (graphical, XML based and π-calculus) and the respective mappings between them. The task model itself is not completely

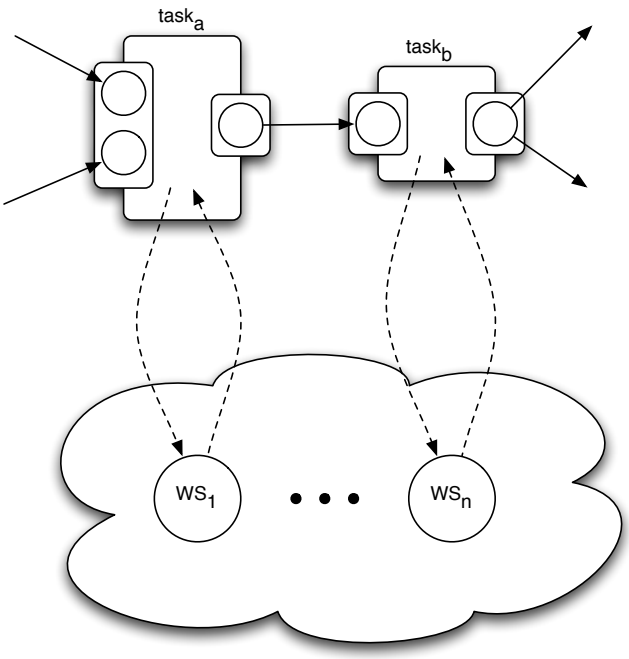


Figure 3.1: Fragment of Graphical Task Model Notation

novel; the model presented here is based on the OpenFLOW task model [2] but it has been extended with other types of task to align it with Web Services and a formal model has been developed to allow verification of workflows described in this notation.

3.2 Task Model

The Task Model is used to define the structure of workflows that correspond to a business process. It is a model of how workflows can be represented that has certain properties and semantics. There are three ways of representing workflows in the Task Model. Firstly, there is a graphical notation, a small example of which is shown in Figure 3.1. This is the representation that is used to design the workflow and depict it's structure. Secondly there is an XML based notation of the Task Model. This is an executable language that can be interpreted by the DECS enactment engine (see Chapter 5 for details on DECS). Finally, it is possible to represent the Task Model using π-calculus which allows formal verification of the workflow against safety and liveness constraints. This section will describe the Task Model showing the graphical representation and then move on to show the formal semantics of the π-calculus representation. The XML representation is primarily used for execution and does not present any novel contribution to this thesis. The XML Schema for the Task Model is included in the Appendix should the reader be interested.

The Task Model is structured in terms of tasks whose execution is controlled through dependen-

Figure 3.2: Overview of Local and Remote WSDL Interfaces

cies. Referring to Figure 3.1, two tasks are depicted, task_a and task_b. Each task in the task model can be considered to be the invocation of a Web Service (we will discuss later about interacting with other technologies), and can see in the diagram that task_a 'invokes' Web Service WS_1 and task_b 'invokes' Web Service WS_n. Each task has an input set and output set associated with it and these are drawn on the left and right hand side of the task respectively. Each set is made up of a number of 'parts', shown as circles within each set. The circles represent individual data items within the set and data dependencies, shown as solid lines in the diagram, are used to control the routing of this data through a workflow. The dotted lines represent messages sent to and received from a web service. They are not temporal dependencies which we will introduce later on and are normally depicted with broken lines.

The Task model is fully aligned with, and takes a large amount of input from the WSDL [165] specification. In particular, WSDL defines four message exchange patterns (MEPs) which dictate the messaging semantics of the web service. These MEPs are request-response, notification, one-way and solicit-response. Each of these will be dealt with in turn below, referring when relevant to the fragments of WSDL shown in Figure 3.3. The listing shows three messages and two portTypes making use of the message definitions. The two port types represent the WSDL definitions exposed by the workflow engine (which is itself a service) and the remote web service. For sake of brevity, these are referred to as local WSDL for the workflow engine and remote WSDL for the web service at the other end of the interaction, as shown in Figure 3.2. It should be noted that at present the Task Model uses the MEPs from WSDL 1.1. In the future we have plans to integrate the additional MEPs that appear in the WSDL 2.0 specification, a task that we do not anticipate being problematic.

3.2.1 Request-Response Task

A request-response task is the primary way to perform a unit of work in the task model. It corresponds to performing a request-response interaction with a remote web service. The task is defined in terms of the WSDL document describing the interface to the remote service as shown in Figure 3.3 and the graphical representation of this is shown in Figure 3.4. The square on the left hand side, represents the input set of the task (named msgA), which contains one input part (named body). These names match the WSDL message and WSDL part for the input of the web service. When the

```
<message name="msgA">
  <part name="body" element="xs:String"/>
</message>
<message name="msgB">
  <part name="body" element="xs:String"/>
</message>
<message name="faultMsg">
  <part name="body" element="xs:String"/>
</message>

<portType name="exampleLocalPT">
  <!-- notification -->
  <operation name="send">
    <output message="tns:msgA"/>
  </operation>

  <!-- one-way -->
  <operation name="receive">
    <input message="tns:msgB"/>
  </operation>

  <!-- solicit-response -->
  <operation name="solRes">
    <output message="tns:msgA"/>
    <input message="tns:msgB/>
  </operation>

  <!-- request-response -->
  <operation name="reqRes">
    <input message="tns:msgA"/>
    <output message="tns:msgB"/>
  </operation>
</portType>

<portType name="exampleRemotePT">
  <!-- request-response -->
  <operation name="reqRes">
    <input message="tns:msgA"/>
    <output message="tns:msgB"/>
  </operation>

  <!-- request-response with fault -->
  <operation name="reqResWithFault">
    <input message="tns:msgA"/>
    <output message="tns:msgB"/>
    <fault message="tns:faultMsg"/>
  </operation>
</portType>
```

Figure 3.3: Abstract WSDL

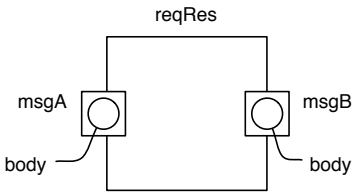


Figure 3.4: Request-response Task

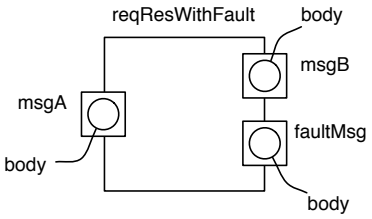


Figure 3.5: Request-response with Faults Task

task's input set is available, a SOAP message representing this is built and sent to the service being represented. The data that comprises this input may come from the output of another task. This is known as a dependency and is described in Section 3.2.8. As this is a request-response interaction, the task blocks until the service returns a response message. This result message is received and maps onto the task's output set (shown on the right hand edge of the task). The syntax for this is the same as the input set, with the circle representing a WSDL part and the square representing the WSDL message. The contents of this message are the output set of the task. When the result set is available it is propagated along any dependencies which are present for that set (described in Section 3.2.8).

Should the WSDL interface to the remote service indicate that the service may return a fault instead of a response message, the structure of the task is different. Fault sets for the task are used to represent the fault message and are shown on the right hand edge of the task along with the output set. The example in Figure 3.5 has one output message and one fault message. These are mutually exclusive as the service will send back either a response message or a fault message. These faults are only intended to capture those faults that are explicitly listed the WSDL document that describes the service, not any messages that are semantically fault messages.

3.2.2 Notification Task

A Notification task is used to represent a one-way send of a SOAP message to a web service. When considered with WSDL, this maps to a one-way MEP of the remote web service (a "one-way" WSDL MEP receives a message). The discrepancy in the name of the task is due to the fact that the WSDL used to define the task is the remote one. As the WSDL interface of interest is describing the remote service receiving a message, the task which executes what can be considered to be the other end of

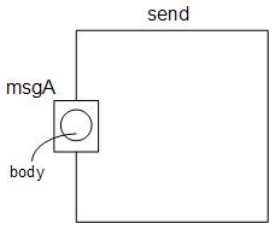


Figure 3.6: Notification Task

this is sending a message. WSDL descriptions tend to only define the messages that an endpoint will consume rather than also defining the messages which will be emitted (except for the latter half of a request-response interaction). If such notification MEPs are defined in the local WSDL document, then this task can be mapped to a notification MEP in that local WSDL document. In practice, the separation is purely pragmatic as one definition will be the inverse of the other.

A notification task is shown in Fig. 3.6 and the syntax is similar to that described for request-response tasks. That is, there is an input set on the left hand side of the task, containing one part. This corresponds to the input message and part in the remote WSDL (or output in local WSDL). Once the input set is complete, the task can be executed and a SOAP message will be sent to the web service. There is no output set associated with this style of task (or fault set) as this is a one-way interaction. As there is no output set, this task is non-blocking, that is the SOAP message will be deposited on a queue and the task will not wait for any response.

3.2.3 Receive Task

A receive task (as shown in Figure 3.7) models the receipt of a SOAP message and maps onto a on a one-way WSDL interaction defined in the local WSDL. The task has no input set as there is no message which will be sent to the remote service. Only an output set is present, shown on the right hand edge of the task and with the same syntax as described previously. The square represents the output set with the inner circle representing the output part. These correspond to the WSDL message and parts which are defined in the one-way interaction in the local WSDL. As there is no input set for a receive task, no data is required to be able to execute the task. However, to control the execution, temporal dependencies as described in Section 3.2.9 can be used. The Task Model is agnostic about addressing and correlation details for the messages. It is clear that from an enactment perspective the message received by the workflow engine for a receive task must be routed to the correct task in the correct instance of the running workflow. This issue will be discussed in Chapter 5 and a solution using WS-Addressing will be presented.

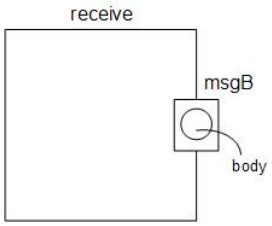


Figure 3.7: Receive Task

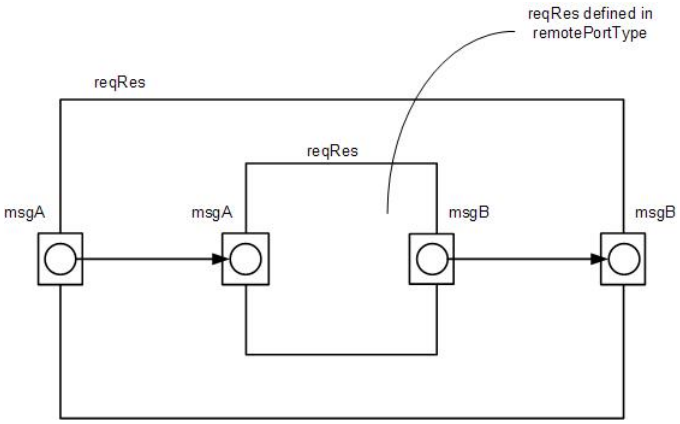


Figure 3.8: An Early Instantiated Process

3.2.4 Processes

Processes have a number of uses within the task model. Firstly, they are the principle way of providing abstraction and re-use of workflows. Secondly, they are used to model the remaining WSDL interaction pattern. Finally, they can be used to implement a recursive action.

A process, as shown in Figure 3.8 represents a larger granularity unit of work than that of a single web service interaction. A process has the same external structure as a request-response style task but instead of representing a WSDL request-response pattern, it represents a process that has an internal structure. This internal structure is also in terms of the task model and can contain all of the types of tasks mentioned previously. This is a way of providing abstraction and re-use as designers of workflows can create a process and then use it within other processes and workflows. In such cases the process can either be considered a black box with an input and output set or may be "exploded" to show its internal structure.

The external structure of the process may be optionally exposed as a web service itself and in this case it matches the remaining WSDL interaction pattern. From a local perspective, this is a request-response interaction whereas from a remote perspective it is a solicit response style interaction. It is the former, or local definition that is used to define the interface to the process. The input set to the process is the WSDL input message from the local, request-response operation and the input parts are the parts which form that message. The output and optional fault sets match those output and

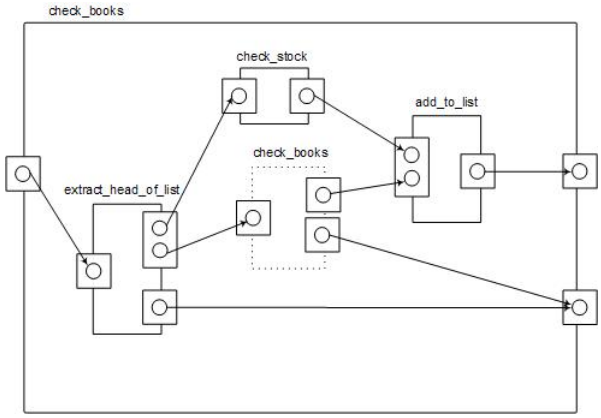


Figure 3.9: A Late Instantiated Process

fault messages defined in the WSDL document and their respective parts.

The final use of a process is as a method of performing recursion. This is achieved by having two logical times when the structure of a process is evaluated and loaded into the execution environment. Under normal circumstances the structure of a process will be loaded when the outer containing workflow is loaded into the execution environment. In such cases, the process can be considered as a form of macro expansion which is performed at the beginning of the execution. This makes monitoring more straight forward as the complete structure of the process is known a-priori. In order to perform recursion, a specialised version of a process is available which is known as a late-instantiated process. The internal structure of such processes is not loaded into the execution environment until the process is able to execute. This is the same time that a SOAP message would be sent to a remote web service if the process were instead a task. Using this mechanism, it is possible for a late-instantiated process to include a copy of itself internally and load a new copy of the process, performing recursion under certain conditions. An example of such a process is shown in Figure 3.9 and is depicted by a dotted outer rectangle rather than a solid one which represents an early-instantiated process. In the example given, recursion is used to process the items on a list. Each item, which in this case is a book, must be checked to see if it is in stock and added to a list if it is. The structure of the process is such that the head of the list is removed and the remainder of the list is passed to another instance of this process. This continues until the list is empty, while in parallel, the check_stock operation is invoked. When the list is empty the extract_head_of_list returns a fault message and the recursion stops. When returning up the chain as each request-response interaction completes, the books are added to the list if they are in stock. It should be noted that the check_stock operation does not return a fault message if it fails, but the add_to_list operation will only add it if the payload of the message from the check_stock indicates success. Further uses of late-instantiated are for performing runtime adaptation and catering for cases where the complete workflow structure is not known in advance. This is described in Section 3.7.

3.2.5 Representing incomplete WSDL and modelling multiple message parts

Although the task model is aligned with the WSDL model for representing web service interfaces, it makes use of a mixture of local and remote WSDL definitions. This is partially because of conventions used in WSDL documents and partially as it makes logical sense to think of certain tasks in certain ways. WSDL has its roots in IDL and has frequently been referred to as "IDL with angle brackets". The conventions used in IDL have found their way into WSDL in that only the operations which consume messages are defined in the WSDL document. This is the same as defining a set of method signatures in IDL. IDL does not define the external methods which are used by the object being described by the IDL. In the same fashion, WSDL does not typically describe the operations of other web services which are invoked from the service being described. This is not a necessity, purely a convention. It is possible to define operations which emit messages in WSDL using either a notification or solicit-response style operation. If this were consistently done it would be possible to define the task model purely in terms of either the local or the remote WSDL document. However, even when complete WSDL (that defines all messages/operations the service emits as well as consumes) is provided it is not natural to describe tasks in this manner. For instance, Section 3.2.1 described the request-response style task in terms of the remote WSDL document. In a full local WSDL description, the equivalent task would manifest itself as a solicit-response operation. Although what is actually happening is performing a solicit-response interaction it is not natural to think in this way. Programmers usually think in terms of "performing an request-response" rather than "initiating a solicit-response".

Most of the examples in the previous sections have followed one of the principles mandated by the WS-I Basic Profile [80], that is, each WSDL message should have one child element named body. Services which adhere to the WS-I Basic Profile are intended to be more interoperable. However, at present not all services do adhere and so it is necessary to be able to be able to interact with those services too. The WSDL shown in Figure 3.10 breaks this rule of WS-I BP by having two messages each with two parts. Fig. 3.11 shows how the task model deals with such messages, with one circle representing each part of the message and thus forming the input/output set of the task. SOAP headers are handled in a similar way in the Task Model. They are treated as a 'part' of the message and are represented graphically as a circle in the corresponding input or output set. They may also be the source or sink on dependencies as described in the following Section.

3.2.6 Representing Tasks that are not web services

Although most of this chapter is concerned with modelling types of web services, it is clear that if Web Services were the only technology available, the task model would be unnecessarily restrictive.

```
<message name="requestOrderStatusMsg">
  <part name="clientId" element="xs:String"/>
  <part name="orderNumber" element="xs:Integer"/>
</message>
<message name="orderStatusResponseMsg">
  <part name="statusCode" element="xs:Integer"/>
  <part name="statusMsg" element="xs:String"/>
</message>
<portType name="exampleOrderStatusPT">
  <operation name="requestOrderStatus">
    <input message="tns:requestOrderStatusMsg"/>
    <output message="tns:orderStatusResponseMsg"/>
  </operation>
</portType>
```

Figure 3.10: WSDL Showing multiple message parts

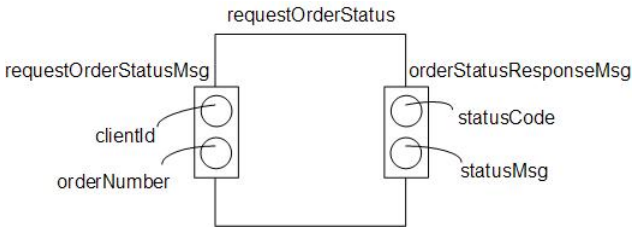


Figure 3.11: A Task with multiple parts

Some actions which are useful within a workflow are too lightweight for Web Services technology to be a sensible choice. For instance, data transformation, logging etc. It is possible to include calls to such applications by using a request response style task. Instead of the task referring to a WSDL interface, the task refers to a Java interface (for example) instead. The input set of the task is the parameters to the method (with one part per parameter), the output set is the return value of the method and optional fault sets are the exceptions which a method may throw. This is similar to the original work on the Task Model where it was applied to Corba technology [170].

The services that are used within a workflow may be out of the control of the designer of the workflow. If this is the case, although the services may perform the required function, the input and output formats may not be perfectly aligned with the data the workflow designer has available. In cases of such mismatches it is possible to introduce lightweight components that transform the data so that it is in the correct format. These components are based on the XML Stylesheet Transformation Language (XSLT) and transform the source document (i.e. message) into the target document using a stylesheet. The idea is similar to that of Shims as implemented in Taverna [171]. The input set for this task contains the source document and the stylesheet and the output set is the target document.

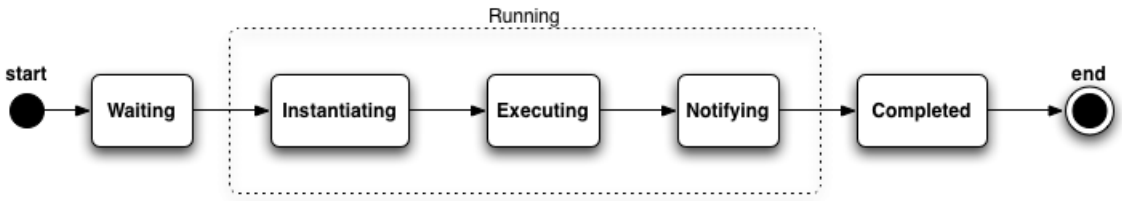


Figure 3.12: Task State Transition Diagram

3.2.7 States of a Task

A task can be in a number of states as shown in Figure 3.12. Initially, all tasks in a workflow are in the waiting state. This means that they have been loaded into the execution environment but do not have the data which comprises their input set yet. Obviously this data will become available at different times for each task, depending on where it is originating from, but when it is available the task moves into a instantiating state. In this phase, different things will occur depending on the structure of the task. If the task is in fact a late instantiated process, the structure will be loaded into the execution environment and will then move to an executing state. If the task is a request-response task, the input set will be used to create a SOAP message to be sent to the remote web service, this will be sent and the task will move into the executing state. For any type of task, at the instantiating phase, if there are dependencies on the start of the execution, these will be fulfilled and data propagated (we will discuss dependencies in the following Section). When the task has finished executing (signalled by the subprocess completing or receiving a SOAP response message), the results are persisted and during the notifying phase, any tasks which are dependant on those results are informed. Finally, the task moves to a completed state with all of its execution data persisted.

It should be noted that a task only moves into the instantiating state once all of its data is available. That is, if there are multiple parts to the input set, the state only becomes instantiating when all of these are available. At this time (when all are available), the version of the data which became available most recently is taken for the task's input set. Once the input set is marked as complete it is never altered, thus creating a persistent record of the input data for that task. In the fields such as bioinformatics this data is known as the 'provenance' of a workflow. This must include all of the data that was used in the workflow to allow it to be repeated at a later date. [172]

3.2.8 Data Dependencies

Data dependencies are used in the task model as the way of routing data from one task to another. They are said to have a *source part* and a *sink part* which correspond to parts in the input/output set of a task (and thus the WSDL message parts). The source and sink of data dependencies can be

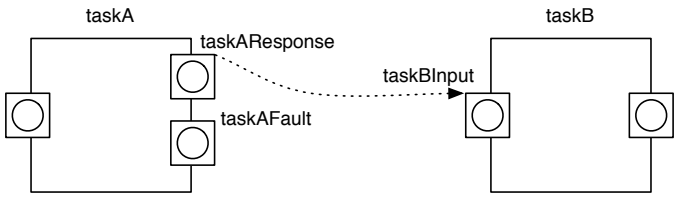


Figure 3.13: Temporal Dependencies

parts in either input or output sets of tasks or processes. The dependency is shown by a solid arrow going from the source part to the sink part (as shown in Figure 3.8).

It is possible to introduce redundant data sources using data dependencies to provide a higher degree of fault-tolerance. For instance, more than one task could be used to generate the same data (say, some data extraction from a bio-informatics database of which multiple copies exist). The tasks which perform this lookup could execute in parallel and the data from the first one to return would be used. This is achieved by having a number of data dependencies with their sources as the output sets of the various tasks querying the data. The sink of all of the data dependencies is the same part of the task which is going to process the data. These are alternatives to each other so the task will execute when the one complete set is available. When a task completes and its output set is available, the data is propagated along all data dependencies which have a source in that output set. If the task at the sink of the data dependency is in either the running or complete state then the data is not copied. Copying the data in this case would lose the provenance, in that the data which formed each input/output set is no longer available: clearly an undesirable situation.

3.2.9 Temporal Dependencies

Temporal dependencies are used in the task model to control the execution order of tasks when there is no data relationship between them. Such dependencies are represented graphically as a dotted arrow between two of the data sets for the tasks rather than between the parts within the data sets. This is because there is no data involved, rather the event in question is the whole data set being available. Thus, if the source of the temporal dependency is an input set, the temporal dependency is considered active when the input set is complete. If the source is an output set then the dependency is considered active when the output set is complete. In the case of a request-response style task, this will be when the web service has returned a response (or fault message). An example is given in Figure 3.13 where a temporal dependency is controlling the execution of TaskB. TaskB can only execute if TaskA completes with the taskAResponse output set. If TaskA completes with TaskAFault, TaskB will not execute.

Temporal Dependencies can be combined into groups to give a finer grained control of the execution of tasks. One Temporal Dependency in each group must be active for the sink task to execute.

Each temporal dependency must belong to exactly one group. This allows for various combinations of dependencies to be used to precisely control when a task should execute.

3.3 Pattern Based Analysis of the Representational Capacity of the Task Model

Analysis presented in [100] has identified a number of recurring structures in both business and scientific workflows. These structures have been distilled into their simplest form and are known as *workflow patterns.* The 21 workflow patterns can be grouped into 6 categories:

- Basic control patterns such as sequence and parallel. (B)
- Advanced Branching and Synchronisation such as merging with and without synchronisation. (A)
- Structural patterns such as arbitrary cycles. (S)
- Patterns involving Multiple Instances of an activity, where the number of instances may or may not be known a priori. (MI)
- State Based patterns such as interleaved parallel routing where activities may execute in any order but never in parallel. (SB)
- Cancellation patterns where certain activities or the whole workflow are explicitly disabled. (C)

The patterns which are explicitly supported by the Task Model are shown in Table 3.1. In the table, + indicates that there is direct support for that pattern in the Task Model, - indicates that there is no direct support and +/- indicates that although there is no direct support it is relatively straight forward to model the pattern using a combination of other constructs (without resorting to providing hand crafted activities/tasks).

The Task Model is able to support all of the Basic Control patterns such as performing activities in sequence and parallel etc. Such patterns are simple and supported by most workflow languages. Any language which did not provide such basic patterns would be at best unpopular and at worst unusable. When considering the more complex patterns it is interesting to investigate why some and not others are supported by the task model and how the task model compares to other workflow languages.

The multi-choice pattern (6) which falls into the Advanced Branching and Synchronisation group is not directly supported by the Task Model. This is due to the fact that the Task Model is developed as a control flow language and the underlying formalisation does not support the required conditional

		Pattern	Task Model	BPEL
1	B	Sequence	+	+
2	B	Parallel Split	+	+
3	B	Synchronisation	+	+
4	B	Exclusive Choice	+	+
5	B	Simple Merge	+	+
6	A	Multiple Choice	+/-	+
7	A	Synchronising Merge	+/-	+
8	A	Multiple Merge	-	-
9	A	Discriminator	+	-
10	A	N out of M Join	+/-	-
11	S	Arbitrary Cycles	-	-
12	S	Implicit Termination	+	+
13	MI	MI without Synchronisation	+/-	+
14	MI	MI with a priori known design time knowledge	+	+
15	MI	MI with a priori known runtime knowledge	+	-
16	MI	MI with no a priori runtime knowledge	-	-
17	SB	Deferred Choice	+/-	+
18	SB	Interleaved Parallel Routing	-	+/-
19	SB	Milestone	-	-
20	C	Cancel Activity	-	+
21	C	Cancel Case	-	+

Table 3.1: Support for Workflow Patterns in the Task Model

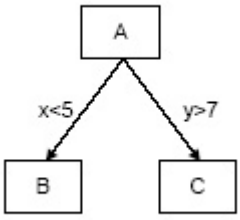


Figure 3.14: Multi-choice Pattern

statements necessary to implement this pattern. The pattern is described as "A point in the workflow process where, based on a decision or workflow control data, a number of branches are chosen". For instance, as shown in Figure 3.14 After performing task A, if x <5 perform task B and if x >= 7 perform task C. In this case, either B or C, or both, or neither could be executed. To be able to model this pattern directly it is necessary to have conditionals placed on dependencies which the Task Model does not contain. The reason for the absence of such conditionals is the difficulty of performing formal analysis if they are present.

Although it is not possible to directly model the multi-choice pattern, it is possible to include activities in a workflow which does model it. Instead of having conditional dependencies, it is possible to add a task which performs the evaluation of the conditional (as shown in Figure 3.15).

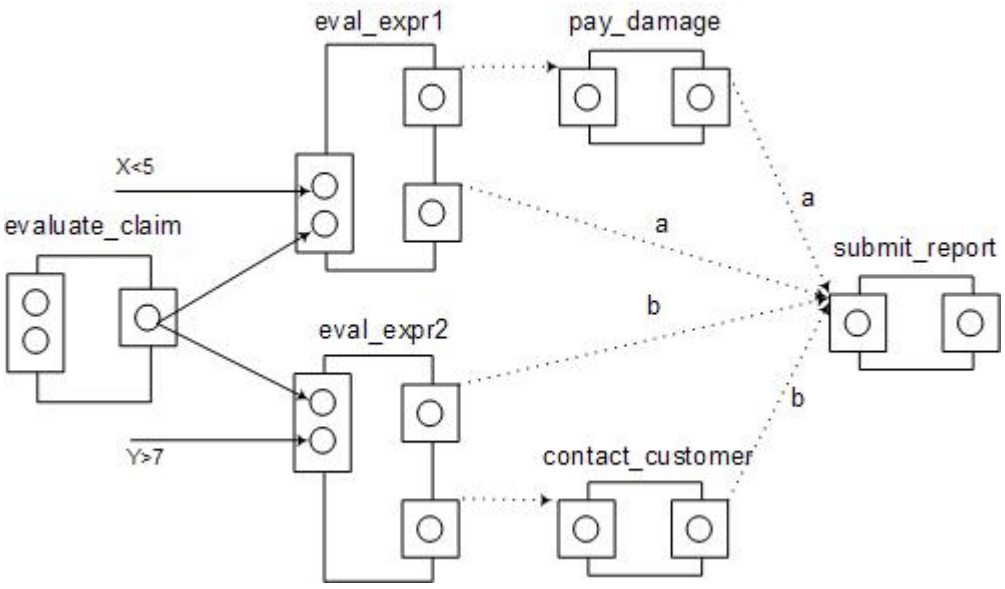


Figure 3.15: Implementation of a Synchronising Merge Pattern

The advantage of this is that it forces the designer of the workflow to deal with all interleavings of the conditional expressions they include making formal analysis more straightforward. Although this is necessary for formal analysis, it is not necessarily sufficient.

If this workaround is used, it is possible to implement the synchronising merge pattern as shown in Figure 3.15. This pattern is used to control the merging after a multi-choice and is defined by Van der Aalst as:

> A point in the workflow process where multiple paths converge into one single thread. If more than one path is taken, synchronization of the active threads needs to take place. If only one path is taken, the alternative branches should re-converge without synchronization. It is an assumption of this pattern that a branch that has already been activated, cannot be activated again while the merge is still waiting for other branches to complete.

The Task Model can use the OR semantics of temporal dependencies to achieve this as the number of paths which must be synchronised is known at design time. In Figure 3.15 at least one of the a temporal dependencies and b temporal dependencies will be fulfilled which will allow the task submit_report to execute. Although this pattern is directly supported in the Task Model, it is only marked as +/- as it requires a workaround in a another pattern to be applicable.

The Multiple Instance patterns cover cases where an activity needs to be instantiated multiple times within a workflow. The various patterns in the group cover the cases where the number of instances is discovered at varying times. It is trivial to implement the case where the number of instances is known at design time as the designer can manually replicate the activity the desired

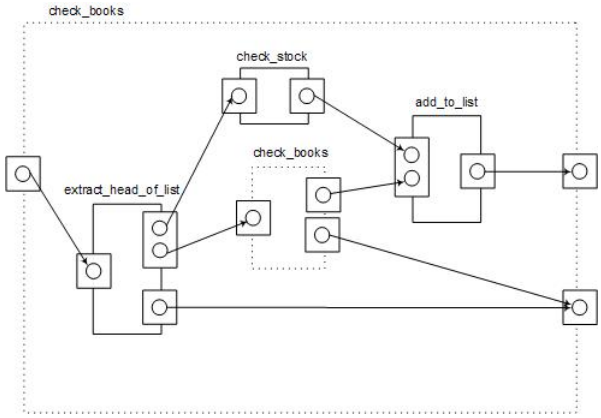


Figure 3.16: Implementation of the MI with a priori runtime knowledge

number of times. If the number of instances is discovered at runtime (pattern 15) the Task Model is able to implement this too. The pattern is realised using a late instantiated task, as shown in Figure 3.16 where the check_books task refers to an instance of itself to perform recursion. As noted in [173] there is no pattern which explicitly targets or describes recursive constructs in workflow languages.

The State Based Patterns are primarily used by petri-net based languages which have an explicit notion of "state". As the Task Model attempts to minimise the amount of state within a workflow it is not possible to implement many of the State Based Patterns. For instance, the Milestone pattern requires a workflow to be in a particular state to allow an activity to be executed. This state is usually modelled by tokens in a petri-net being in certain positions. As the Task Model has no way of describing such state it is not possible to model this pattern. It is possible to model the deferred choice pattern in the Task Model. This is because it relies on an external event which can be captured by a message sent to the workflow and received by a receive task.

It is not possible to implement the Cancellation Patterns using the Task Model. This is because there is no notion of cancellation at the language level. However, as explained later in this chapter, it would be possible to dynamically reconfigure a workflow such that it had the effect of cancelling a workflow that is currently executing.

The patterns that BPEL is able to support are also shown in Table 3.1 as a comparison. This data has been taken from [174] and a number of other workflow languages have been analysed using the patterns [100]. We can see that the Task Model and BPEL are very different languages, with the Task Model able to represent patterns such as Discriminator and N out of M join which BPEL cannot. Conversely BPEL can represent the cancellation patterns that the Task Model cannot.

One of the differences between the Task Model and BPEL is the amount of state held by the language. BPEL holds a lot of state using what are essentially global variables and tasks can depend on variables being in a particular state before they execute. There is no equivalent notion in the

Task Model and it can be considered more of a data flow language with BPEL more of a control flow language that handles data flow using global variables. BPEL has looping constructs and conditionals which depend on the state of variables, again concepts that are not present in the Task Model. As BPEL relies on the use of global variables to control the looping and conditionals it is non-trivial to perform decentralised execution as the state of the whole process must be replicated across each enactment engine. In the Task Model each task is encapsulated with its own state meaning that it is possible for different nodes to control the execution of that task.

BPEL was created from a fusion of IBM's Web Services Flow Language (WSFL) and Microsoft's XLANG. As such, it inherits the block structured nature of XLANG as well as the directed graphs of WSFL. This combination means that there are often multiple ways of expressing the same control structure making the language more complex for both the end user and for the purposes of analysis. For instance, the specification of a sequence of activities can be realised either through the use of the `sequence` element or a `flow` element. In the latter case links are used to enforce a particular order on parallel elements. There are subtle restrictions on the use of links to ensure that they do not cross scope boundaries or create cycles. The results of this is that although BPEL is able to model the patterns shown in Table 3.1 it is not trivial to combine the patterns into a complete process as some make use of the block structured elements and some the directed graph elements.

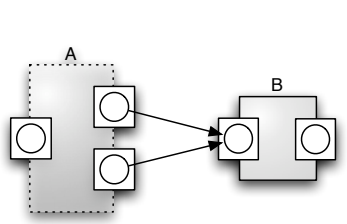
3.4 Formalising the Task Model

3.4.1 Motivation

The motivation behind building the Task Model on a formal notation is that it allows verification that certain properties, i.e. safety and liveness conditions [175] are always present. To elaborate on this, there are a number of undesirable situations which could occur within a workflow as shown in Figure 3.17 and Figure 3.18:

- Unreachable tasks: Execution is controlled by multiple dependencies which are explicit alternatives to each other so task can never execute
- Cyclic dependencies: Execution of a task depends on the output of the same task (may be explicit or implicit)

In addition to the problems identified above which can be classed as structural problems with workflows it is also possible to distill application specific problems which we want to ensure do not occur. Such 'semantic' problems are heavily dependent on the application domain but examples might include: 'the customer never receives the goods if payment has not succeeded' or 'the customer always receives the goods if the payment succeeds'. The formalisation of the workflow allows us to



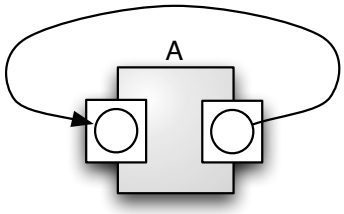


Figure 3.17: Unreachable Tasks Figure 3.18: Cyclic Dependencies

verify that such assertions are true or false. It should be noted that there are some types of error that can occur within a workflow that we are not attempting to detect, for example, infinite recursion.

When choosing how to formalise the Task Model there were a number of languages that were considered. CCS [176], CSP [177] and Petri-nets [178] are all popular formalisms for describing concurrent systems and workflow systems in particular. We chose π-calculus for a number of reasons but principally for the inbuilt notion of 'mobility'. We will see this used in Chapter 4 to describe dynamic protocols where not all parties are known at the beginning of the protocol. We felt that this offered a significant advantage over other protocol descriptions and was able to describe certain protocols precisely. Although the Task Model formalism does not require mobility, using the same formalism removes the possibility of an impedance mismatch between formal models of the Task Model and protocol descriptions presented in the following Chapter.

3.4.2 How the task model maps onto π-calculus

To allow reasoning about workflows with respect to safety and liveness properties, the Task Model has a formal basis in the π-calculus [154]. We chose the π-calculus as it has been shown to be suitable in the Service Oriented arena [21] and [99]. We do not exploit the full power of the π-calculus in our formalisation of the task model but we do become dependant on channel passing once we introduce the roles which will be shown in Chapter 4. Also, weak open bi-simulation is a convenient way of reasoning about the workflows that we describe in the Task Model notation.

It is possible to translate from the XML format of the language to the π-calculus format. In the π-calculus format, tasks are represented as π-calculus processes, and dependencies linking the tasks, represented by π-calculus channels. To represent a dependency being fulfilled a name is sent along a channel. This is the same for both data and temporal dependencies as the latter can be considererd equivalent to the former if the contents of the data are ignored; it is simply a signal. Each of the tasks in the Task Model represents the invocation of a Web Service, we will see in Section 3.5.1 how this is modelled but here we represent the invocation as an internal action (τ_{interact}).

Formally, a workflow is represented by the tuple (T, TDG, TD, DDG,DD, DS) such that

- T is the set of Tasks
- TDG is the set of Temporal Dependency Groups
- TD is the set of Temporal Dependencies
- DDG is the set of Data Dependency Groups
- DD is the set of Data Dependencies
- DS is the set of Data Sets (input sets/outout sets/fault sets)

We formalise a task, named A as follows[2]:

$$A = (\nu\ iwt,\ iwd,\ nit,\ nid,\ owt,\ owd,\ not,\ nod) \tag{3.1}$$

$$(iwt \mid iwd).(nit \mid nid).\tau_{interact}.(owt \mid owd).(not \mid nod) \tag{3.2}$$

$$\mid InputWaitTemp(iwt) \mid InputWaitData(iwd) \tag{3.3}$$

$$\mid NotifyInputTemporal(nit) \mid NotifyInputData(nid) \tag{3.4}$$

$$\mid OutputWaitTemporal(owt) \mid OutputWaitData(owd) \tag{3.5}$$

$$\mid NotifyOutputTemporal(nod) \mid NotifyOutputData(nod) \tag{3.6}$$

$$\tag{3.7}$$

Line 3.2 shows the basic, top level control flow for the task. There are five distinct phases of the task and these represent the states of the task that were shown in Figure 3.12. Initiatlly when the task is in the waiting state it is waiting for the input data to become available and the temporal dependencies that define the control flow to be enabled. These are formalised using the iwt and iwd channels respectively. The way in which these channels are used will be described below, but once the dependencies are fullfilled a message will be sent along the iwt and iwd channels and A will progress into the instantiating state. In the instantiating state the task must notify any downstream tasks which have dependencies on its input set. The completion of this is signalled by empty messages being sent along the nit and nid channels, allowing the task to progress into the executing state. As it is the control flow of the workflow that we are interested in modelling at this stage we model the execution of the task as an internal action using $\tau_{interact}$. In later stages we will show the interaction with the Web Services that the task is representing. If A is a composite task then there will be dependencies on the output sets indicating how they are built and what temporal conditions must be met for the task to complete. This is controlled by the owt and owd channels which will

[2]With respect to readability, the names of channels defined in this process are abbreviated names of the processes they are passed to. For instance, iwt corresponds with InputWaitTemp and nid corresponds with NotifyInputData etc.

receive an empty message when the relevant dependencies are fulfilled. Finally, before A can move to the complete state it must notify any downstream tasks which have dependencies on its output sets. When this has been done messages will be received on the not and nod channels, allowing A to complete. In each of the states defined above, actions for the temproal and data dependencies are being carried out in parallel. This is important otherwise deadlocks might be introduced into the model which do not exist in practice due to the blocking nature of π-calculus message passing. Lines 3.3 to 3.6 show parametric calls to processes which we will define below to do either wait on upstream tasks or notify downstream tasks of the state of the task. Each of the processes is executed in parallel with the part of A described above and is passed a channel as a parameter. This channel will be used to signal when that part of the task has completed.

To model the task waiting for its temporal and data dependencies to be fulfilled we introduce the following processes:

$$InputWaitTemporal(r) = (\nu\ f)\ \Pi_{i \in G}(temp_i.\overline{f}.(!temp_i)) \mid \{f\}_{j=|G|}.\overline{r} \tag{3.8}$$

$$InputWaitData(r) = (\nu\ f)\ \Pi_{i \in H}(part_i.\overline{f}.(!part_i)) \mid \{f\}_{j=|H|}.\overline{r} \tag{3.9}$$

where

$$G = \{x\ \in\ TDG,\ y \in\ TD \mid y\ \in\ x\ \wedge\ sink(y)\ =\ A\ \wedge\ sinkType(y)\ =\ input\}$$

$$H = \{x\ \in\ DDG,\ y \in\ DD \mid y \in\ x \wedge\ sink(y)\ =\ A\ \wedge\ sinkType(y)\ =\ input\}$$

The two processes have the same structure except that they operate over temporal and data dependencies respectively. The processes initially restrict the scope of a new name, f to that process. There are in fact two parts of each process executing in parallel and the channel f will allow them to communicate without interference from other processes. This is a technique that we shall see repeatedly in this formalisation. The first part of the process $\Pi_{i \in G}(temp_i.\overline{f}\ldots$ executes a number of receive actions in parallel, one on each of the items that belongs to the set, G, and then sends a message on the private channel, f. The set G contains one item for each of the distinct temporal dependency groups that the input of A is a sink of. If we recall the description of temporal dependencies and their groupings: each dependency within a group is an alternative for each other; that is one dependency in each group of temporal dependencies must be fulfilled for the task to execute. This matches the formalisation described above where we wait for a message on a channel named $temp_i$ which represents that dependency group.

Waiting for messages on these dependency groups presents us with a problem: we are only interested on the first message that is received on each group. There may be many processes that

can send a message on the $temp_i$ channel and we must discard all the other messages. To achieve this we use the replication operator (!) which will initiate an infinite number of receiving prefixes on the $temp_i$ channel, acting as a garbage collector. The second half of the InputWaitTemporal process is trying to solve the converse problem to that just described: we explicitly want to wait for a certain number of messages on a channel before continuing. We must receive a message from each of the dependency groups in order for the task to be executed. This is achieved by having sequential receiving prefixes on the channel f, one for each of the cardinality of the set, G. Finally, when all of these have been received we sent an empty message on the channel r (although this is actually the channel iwt due to renaming).

When considering the naming of the data dependency aspect of this process it might seem strange that the name $part_i$ has been used. However, this becomes more natural when you consider the semantics of input sets in a similar way that we used to think about the temporal dependency groups. The input set for a task is comprised of a number of 'parts'. Each of these parts is the sink of a dependency which describes 'where the data comes from'. However, there may be a number of data dependencies, any one of which could supply the data needed to execute the task. These dependencies form a dependency group, therefore one part is the sink of many dependencies within one group. The way that we model this is to have a channel that represents that part, namely $part_i$. Only one process can receive messages on that channel, but many processes can send messages along the channel - any process that has a dependency who's sink is that part.

In order to notify downstream processes of A's input data, when the dependencies have been fulfilled, we introduce two more processes:

$$NotifyInputTemporal(r) = (\nu\ f)\ \Pi_{i \in J}(\overline{temp}_i.\overline{f}) \mid \{f\}_{j=|J|}.\overline{r} \tag{3.10}$$

$$NotifyInputData(r) = (\nu\ f)\ \Pi_{i \in K}(\overline{part}_i.\overline{f}) \mid \{f\}_{j=|K|}.\overline{r} \tag{3.11}$$

where

$$J = \{x\ \in\ TDG,\ y\ \in\ TD \mid y\ \in x\ \wedge\ source(y)\ =\ A\ \wedge\ sourceType(y)\ =\ input\}$$

$$K = \{x \in\ DDG,\ y\ \in\ DD \mid x \in\ y \wedge\ sourceTask(y)\ =\ A\ \wedge\ sourceType(y)\ =\ input\}$$

The structure of the processes that fulfill the dependencies on the input data is very similar to that described earlier. However, it differs in a few key areas. Firstly, the parallel execution of the prefixes involves sending empty messages on $temp_i$ and $part_i$ respectively. Secondly, there is no need to have the replicated garbarge collection. This is because we want to send a message along each of the channels that represents a dependency even if it will not have any effect. The technique used to

communicate between the different processes executing in parallel and ensure all messages are sent prior to completion is the same as presented before. The sets J and K that are being acted upon are very similar but deal with dependencies that have their source on the input of A rather than A as the sink.

$$OutputWaitTemporal(r) = \Pi_{i \in M}(\nu\ f)\big(\Pi_{j \in N}(temp_j.\overline{f}) \mid \tag{3.12}$$

$$\{f\}_{k=|N|}.\overline{done}\big) \mid \tag{3.13}$$

$$done.(\overline{r} \mid !done) \tag{3.14}$$

$$OutputWaitData(r) = \Pi_{i \in M}(\nu\ f)\big(\Pi_{j \in P}(part_j.\overline{f}) \mid \tag{3.15}$$

$$\{f\}_{k=|P|}.\overline{done}\big) \mid \tag{3.16}$$

$$done(\overline{r} \mid !done) \tag{3.17}$$

$$\tag{3.18}$$

where

$$M = \{x \in DS \mid (type(x) = output \vee type(x) = fault) \wedge task(x) = A\}$$

$$N = \{x \in TDG,\ y \in TD \mid y \in x \wedge sinkgroup(y) = m\}$$

$$P = \{x \in DDG, y \in DD \mid y \in x \wedge sinkgroup(y) = m\}$$

The OutputWaitTemporal and OutputWaitData processes are used to ensure that the dependencies on any output sets for the task are fulfilled before the task completes. The common situations to have dependencies on the output of a task are for composite tasks, a.k.a. processes or for one-way receive tasks where no message is sent (hence no input set) but one is received. A task in the Task Model may have multiple output sets and once one is complete the task may complete. Each output set may have multiple parts and multiple temporal dependencies. It is more complicated to model this scenario than the one for input sets where there is a single input set that must be completed. In order to do so we introduce an extra level of parallelisation and an extra process to help with the synchronisation.

For each output set we create a parallel process and also a private channel f for them to communicate over using the $\Pi_{i \in M}(\nu f)\ldots$ notation. Each of these processes corresponding to output sets will in turn spawn a parallel process for each of the temporal dependencies whose sinks are that set. This is similar to a double nested loop of imperative programming languages, where all of the inner loops spawn new threads. Each of these inner processes listens on a channel $temp_j$ which will indicate that dependency is fulfilled. When this occurs they send a message along the channel f

which is private to that output set. A number of parallel process exist which are listening to the private f channels as shown on line 3.13. Each of these is waiting for N names to be sent along the channel where N is the number of temporal dependency groups that exist on that output set (the cardinality of the set). When the final one of these has been received that indicates that the output set is complete and the task can complete. To indicate this a message is sent over a channel named *done* which is common to all the processes. A process executing in parallel (shown on line 3.14 receives an empty message on this channel and signals to A using the channel r received by the parametric call. Finally, as with the input sets we use a replicated receive on the done channel (!*done*) to act as a garbage collector. This is necessary as the other output sets may complete at a later stage and the *done* messages must be dealt with. The OutputWaitData process shown on lines 3.15 to 3.17 is similar to OutputWaitTemporal described above except that it deals with messages received on the $part_j$ instead of $temp_j$ to model data dependencies.

If there are no dependencies of a particular type then we must model this differently. If we use the above model then task A will block because it will never receive a message sent over channel r. This is because set N (or P) is empty so nothing will occur in the parallel execution $\Pi_{j \in N} \ldots$. We cannot add a process to the model that explicitly deals with this situation as it might be chosen non-deterministically when there are dependencies that must be fulfilled. This would result in errors in the model as the task might complete execution before it ought to. To overcome these problems we define the following:

$$if\ N\ = \emptyset\ then\ OutputWaitTemporal(r)\ =\ \overline{r} \tag{3.19}$$

$$if\ P\ = \emptyset\ then\ OutputWaitData(r)\ =\ \overline{r} \tag{3.20}$$

In these situations, the processes do nothing, simply signalling to A that it can continue. This is because there are no dependencies that must be fulfilled for the task to complete.

The final part of the formalising of the Task Model deals with notifying downstream tasks that an upstream task, A, has completed with a particular output set.

$$NotifyOutputTemporal(r) = \sum_{i \in M} \left((\nu\ f) \Pi_{j \in Q} (\overline{temp_j}.\overline{f}) \mid \{f\}_{k=|Q|}.\overline{r} \right) \tag{3.21}$$

$$NotifyOutputData(r) = \sum_{i \in M} \left((\nu\ f) \Pi_{j \in R} (\overline{part_j}.\overline{f}) \mid \{f\}_{k=|R|}.\overline{r} \right) \tag{3.22}$$

$$\tag{3.23}$$

where

$$M = \{x \in DS \mid (type(x) = output \vee type(x) = fault) \wedge task(x) = A\}$$

$$Q = \{x \in TDG, y \in TD \mid y \in x \wedge sourcegroup(y) = m\}$$

$$R = \{x \in DDG, y \in DD \mid y \in x \wedge sourcegroup(y) = m\}$$

In order to model the fact that only one of the output sets has been enabled and so only the dependencies from this set should be fulfilled we use a non-deterministic choice. This means that only one of the following processes will be executed and all of the others will be skipped and is shown using $\sum_{i \in M}$. The non-deterministic choice is between sending messages over the dependencies for each output set so this means that the dependencies for the other output sets will remain unfulfilled. The processes that contribute to the non-deterministic choice contain the parallel execution of processes for each temporal dependency ($\Pi_{j \in Q} \ldots$). Each process sends messages over the temporal dependencies that have their source in that output set $\overline{temp_j}$. We use the same technique described earlier to count the number of messages that have been sent and then send a message over r indicating completion. Again the data dependencies are equivalent to the temporal dependencies but with different names.

Again, if there are either no temporal dependencies or data dependencies then we must define special processes. This is for the same reasons outlined above: if we do not then the process will indicate deadlock even when it does not actually occur.

$$if\ Q = \emptyset\ then\ NotifyOutputTemporal(r) = \overline{r} \tag{3.24}$$

$$if\ R = \emptyset\ then\ NotifyOutputData(r) = \overline{r} \tag{3.25}$$

3.5 Example

It will help the description that follows to have an example presented here that can be expanded upon in the forthcoming sections. The scenario has four participants, a buyer, seller, bank and shipping organisation. Each of these is considered distinct and wishes initially to remain autonomous and hard organisational boundaries exist between each organisation. Fig. 3.19 shows the global structure of the interactions between each organisation and the messages exchanged at a very high level. The scenario makes use of the RosettaNet standard Partner Interface Process (PIP) messages for inter-organisational messaging [179]. As defined in Chapter 2, the PIP messages specified by RosettaNet

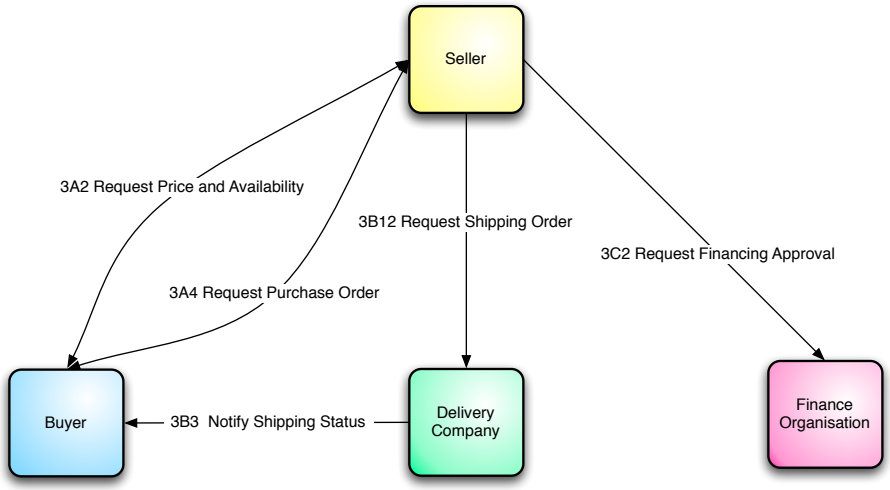


Figure 3.19: Global structure of the scenario

are a set of precisely defined schemas for exchanging business level documents (and elementary semantics and ordering information about these messages). For the purpose of this example, the PIP messages may be considered as any other XML messages which both the sender and receiver understand the syntax and semantics of the message.

The stimulus for the scenario comes from the Buyer who decides the need to purchase something from the Seller (the act of Seller selection and reasons for purchase are out of the scope of the scenario). Initially, the Buyer sends a request for price and availability message (PIP 3A2) to the Seller to determine whether the goods are available. Assuming that the goods are available and the price is acceptable to the Buyer, the Buyer initiates PIP3A4, request purchase order. The Seller processes this message and may communicate with the Bank.

and Shipper using PIPs 3C2 and 3B12 respectively. PIP3C2 requests financing on behalf on the Buyer from the Bank (can be thought of as the Seller processing the Buyer's credit card). Assuming this is successful PIP3B11 arranges shipping of the products and the Shipper will notify the Buyer of the shipping arrangements using PIP3B3. Some of the PIPs described above are request-response style interactions whereas others are one-way interactions. The request-response style are indicated by double headed arrows whereas one-way interactions have only a uni-directional arrow indicating the direction of the message. The workflows which process each of the PIP messages are discussed further in the following sections and more information is given about the PIP messages themselves.

In order to show how a real workflow is formalised using the π-calculus let us consider the workflow for one of the parties. The party we will look at is the Seller as this is the most complicated of the workflows involved. A graphical representation of the workflow is shown in Figure 3.20.

The workflow is fairly straight forward and is used to process a purchase order received from a

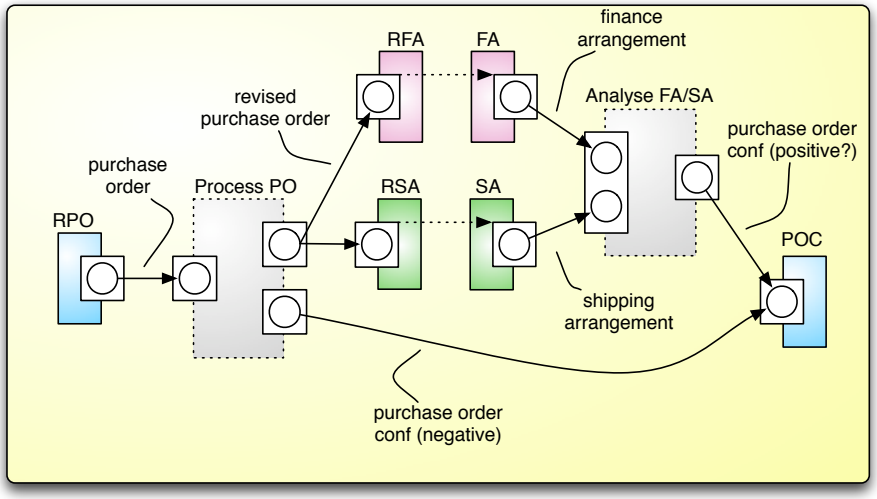


Figure 3.20: Seller's workflow

business partner. Initially a Request for Purchase Order is received by the one-way task RPO. This is passed via a data dependency to the Process PO task. This task is used to analyse the purchase order and is in fact a late instantiated task. This means that the structure of the task will not be loaded until runtime. However, here we are interested in the control flow of the workflow and so we will treat this as a normal task with input and output sets. The Process PO task has two possible output sets, one indicating success and the other failure. The success one is used when the products requested (or a subset thereof) are in stock. The failure output set is used when the products are not in stock. In this latter case there is a data dependency to the Purchse Order Confirmation task that will send a negative confirmation to the Buyer. If the Process PO task is successful the purchase order is passed to two tasks in parallel. The purchase order will have been revised by the Process PO task to include details of the cost of the products ordered and their weight. This allows the two downstream tasks to deal with the financing and the shipping of the goods. The Request Financing Arrangement RFA task is a notification task which sends a message to a Web Service that deals with financing. The Request Shipping Arrangement task is the equivalent to arrange shipping. Following both of these are one-way receive tasks to receive the results of the requests from the other parties. The Analyse FA/SA task analyses the results of the finance and shipping requests. It has two input parts in the same input set and both must be available for the task to execute. There is only one output set from this task but it might contain either a positive or negative purchase order confirmaiton (POC) depending on the contents of the financing and shipping arrangements messages. In either case, this is sent to the Buyer using the one-way POC task.

A full description of the formalisation of this workflow is given in the appendix but we will now show two of the tasks which prove most interesting from a formalisation perspective. Firstly we will

look at the Process PO task which is shown in more detail in Figure 3.22.

3.5.0.1 The Process Purchase Order Task

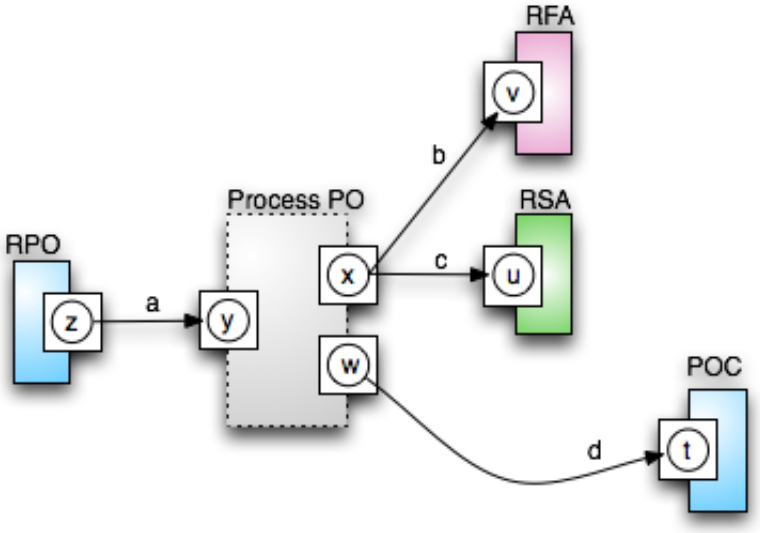


Figure 3.21: Detailed View of the Process PO Task

The ProcessPO task is defined using a parallel composition of the processes we described earlier. This gives the following global process. Later we will describe each of these processes that is of interest.

$$
\begin{aligned}
ProcessPO \; &= (\nu \; iwt,\; iwd,\; nit,\; nid,\; owt,\; owd,\; not,\; nod) && (3.26)\\
&\quad (iwt \mid iwd).(nit \mid nid).\tau_{interact}.(owt \mid owd).(not \mid nod) && (3.27)\\
&\quad \mid InputWaitTemp(iwt) \mid InputWaitData(iwd) && (3.28)\\
&\quad \mid NotifyInputTemporal(nit) \mid NotifyInputData(nid) && (3.29)\\
&\quad \mid OutputWaitTemporal(owt) \mid OutputWaitData(owd) && (3.30)\\
&\quad \mid NotifyOutputTemporal(nod) \mid NotifyOutputData(nod) && (3.31)
\end{aligned}
$$

As can be seen below, most of the processes involved with modelling the Process Purchase Order task are simply reduced to notifications indicating that that aspect of the process is empty and will complete immediately. This is true for all of the processes dealing with temporal dependencies as there are neither any temporal dependencies controlling the exectuion of the task, nor any that control any downstream tasks based on the state of ProcessPO. In addition, there are no data

dependencies that must be notified on the input set for Process PO, nor any that define where the output set is built from as we are treating this as a simple request-response task. Therefore, the only two processes that form any interest are InputWaitData which waits for the data dependencies on the input set to be fulfilled, and NotifyOutputData which propogates the data dependencies on the output set when the task completes.

$$InputWaitTemporal(r) = \overline{r} \tag{3.32}$$

$$InputWaitData(r) = (\nu\ f)\ part_y.\overline{f}.(!part_y) \mid f.\overline{r} \tag{3.33}$$

$$NotifyInputTemporal(r) = \overline{r} \tag{3.34}$$

$$NotifyInputData(r) = \overline{r} \tag{3.35}$$

$$OutputWaitTemporal(r) = \overline{r} \tag{3.36}$$

$$OutputWaitData(r) = \overline{r} \tag{3.37}$$

$$NotifyOutputTemporal(r) = \overline{r} \tag{3.38}$$

$$NotifyOutputData(r) = (\nu\ f)(\overline{part_v}.\overline{f} \mid \overline{part_u}.\overline{f} \mid f.f.\overline{r})\ +\ (\nu\ f)(\overline{part_t}.\overline{f} \mid f.\overline{r}) \tag{3.39}$$

Both of the processes that comprise the ProcessPO task defined above are relatively straightforward. The InputWaitData process consists of two processes executing in parallel. The first process receives an empty message on channel $part_y$. This corresponds to the data dependency being fulfilled. An empty message is then sent on private channel f and a replicated receive on channel $part_y$ is initiated. This replicated receive is used as we do not know if there are any alternatives for this data dependency (in this case there are not, but in the general case there may be). The second process of InputWaitData is used as a counter to count the number of parts that have had dependencies fulfilled. This is achived by listening for messages on channel f as many times as messages are expected. As there is only one part in the input set of ProcessPO the process only needs to wait for one message on channel f. When this message has been received, an empty message is sent on channel r, the channel that was received in the parametric call to the process. This indicates to the higher level ProcessPO process that all of the data dependencies have been fulfilled.

The NotifyOutputData process is responsible for fulfilling the data dependencies that exist on ProcessPO's output sets. It consists of the non-deterministic choice between two processes. These correspond to the two output sets of ProcessPO. The upper output set has two data dependencies emanating from it and the lower set has one. The first half of the choice in the process sends messages along the $part_v$ and $part_u$ channels and then along the private f channels. The messages that are sent along the f channels are received by a process which is running in parallel to the two mentioned previously. Again, this is counting the number of messages sent and when both have been

sent, a message will be sent along the r channel to the ProcessPO process. The second half of the non-deterministic choice is identical to the first half, except there is only one message sent, $part_t$.

3.5.0.2 The Analyse Finance Arrangement and Shipping Arrangement Task

The Analyse Finance Agreement and Shipping Agreement task is different from ones that we've seen before as it has two parts in its input set. This means it must wait for both before executing.

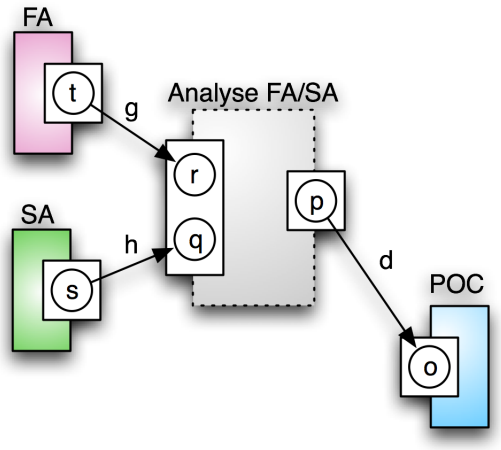


Figure 3.22: Detailed Views of the Analyse FA/SA task

For brevity we ommit the top level formalisation of the task as it is identical to the ProcessPO task described above but with a different name. Also, we ommit all of the processes that only contain notifications sent over r ($\overline{r}$) as they do not add anything to the discussion. With these considerations we are left with the following formalisation of the AFASA task:

$$InputWaitData(r) = (\nu\ f)\ part_r.\overline{f}.(!part_r) \mid part_q.\overline{f}.(!part_q) \mid f.f.\overline{r} \tag{3.40}$$

$$NotifyOutputData(r) = (\nu\ f)\ \overline{part_o}.\overline{f} \mid f.\overline{r} \tag{3.41}$$

As the AFASA task has two input parts it must wait for both of these to become available before it can execute. This is shown on line 3.40 where there are parallel processes which wait for $part_r$ and $part_q$. Although it is not the case here we must still use a replicated receive over these two channels to garbage collect any alternatives for the data dependencies. In the same way as before we count the number of parts that have been received using another parallel process that has sequential receives on private channel f. As there are two parts that comprise the input set there are two receives on channel f. Once both these have been received a message is sent along the extruded channel r

indicating the task may now execute. There is only one output set on the task and only one output dependency emanating from it. Therefore when the task completes there is only one message that must be sent to a downstream task to fulfill the dependencies. This message is sent over the $part_o$ channel and after synchronising on the private f channel the process can notify completion over r.

3.5.1 Analysis of the Workflows

Section 2.5 gave an overview of bi-simulation in the π-calculus. Of the three types of bi-simulation, the weak open bi-simulation of Sangiorgi is the most applicable to us [158]. In this form of bi-simulation non-observable transitions are ignored. This is advantageous as it allows us to focus on certain aspects of interest and consider everything else non-observable to the bi-simulation analysis.

Much of what we are interested in showing by the analysis of the workflow can be considered 'reachability' [180]. If a task is reachable within a workflow it will execute successfully, if it is not reachable it will not execute. We should remember at this point that the defnition of 'executes successfully' is from the perspective of control flow in the workflow, and does not take into account any application behaviour that the task has. If we look back to the diagrams presented in Figure 3.17 and 3.18 we can see that all of the problems that we are trying to detect can be considered a problem in 'reachability':

- Unreachable tasks : Task B cannot be reached as it has two mutually exclusive dependencies
- Cyclic Dependencies : Task A cannot be reached as it depends on its own output.

If we wish to decide whether a certain task is reachable it is necessary to see whether the $\tau_{interact}$ from line 3.2 may be executed. However, by definition, this is an unobservable action but if we replace $\tau_{interact}$ with a send prefix $\overline{interact}$ we can observe this action. If we define the channel *interact* to be the only observable part of the workflow then we can use bi-simulation to compare it to other processes. For example, if we compare it to the process 0 this tells us whether the task is reachable or not: if the workflow is weak open bi-simulation related to 0 then the task in question is not reachable, otherwise it is reachable. This is quite intuitive: The process of weak open bi-simulation reduces the processes according to the π-calculus reaction rules. Clearly the process 0 cannot be reduced any further. As we reduce the workflow according to the reduction rules, weak open bi-simulation ignores the regexes that are formed by the matching sends and receives within the workflow process. By defining an observable part of the workflow in $\overline{interact}$ we introduce something that cannot form a regex with anything internal to the workflow process. During the comparison if this state is reached it will not be possible to progress using the reduction rules. Therefore the workflow will never reduce to the empty process, 0 and so is not weak open bisimular to 0.

By selecting different places to place the observable action we can assert different things about the workflow. For instance, we can place an observable action in the execution stage of a task

as presented above. If we iterate over this changing the task each time we can easily show that every task within the workflow is reachable and therefore there are no unreachable tasks or cyclic dependencies. We might also wish to show that the final task completes execution (all of it's output dependencies are fulfilled and notified). To do this we could place an observable action following the NotifyOutputTemporal and NotifyOutputData have signalled their completion (at the end of line 3.2.

We are not restricted to inserting a single observable action, nor only comparing the workflow to the 0 process. If we want to assert that certain sets of interactions never happen, we can still use bi-simulation. For instance, earlier we mentioned an undesirable series of events where the payment is rejected but the goods are still delivered. Although the workflow we presented earlier does not support checking for this (the arrange finance and arrange shipping occur in parallel), we could imagine one that does. For example we could insert an observable action, $\overline{pay\_cancelled}$, in a task that dealt with the situation where payment had been cancelled. We could also insert an observable, $\overline{deliver}$, action into a task that dispatched the goods to the customer. If we then performed a weak open bi-simulation comparison to the process $\overline{pay\_cancelled}.\overline{deliver}$ we would know whether that series of actions was ever possible. This style of analysis is a powerful tool and we can check for complex scenarios that break business rules. Although tool support for analysing π-calculus models is not as mature as for some other formalisms such as petri-nets, some tools do exist which are able to perform weak open bi-simulation comparisons such as the Mobility Workbench [181] or ABC [182]. In the future we plan to evaluate these tools with the aim of integrating one of them into a graphical tool for designing workflows.

3.6 Scalability of the Task Model

The task model has been defined in such a way that a minimum amount of state must be held to determine which tasks can subsequently be executed. Chapter 5 gives a full description of how various distribution patterns can be used to satisfy the requirements of a Virtual Organisation or reduce data transfer. However, it should be noted that the task model itself does not attempt to describe how a workflow should be enacted. One way to achieve this, and the one that is implemented in DECS is to provide a deployment descriptor which defines which node in a set of enactment engines is responsible for coordinating each task. This allows a separation of concerns when designing workflows so it is possible to consider the application requirements and deployment requirements separately. This also allows the same workflow to be deployed differently by different organisations. This approach is similar to that taken by Van der Aalst in [183] who advocates an inheritance structure for inter-organisational workflows. He proposes three stages of workflow development: design of a global public workflow; partitioning of the public workflow over the participants; each

participant generating a workflow which 'inherits' from their part of the public workflow. We will see in Chapter 5 that we take a similar approach. However, we do not explicitly include stage three of his model, we assume that the workflow is complete before partitioning. There is no reason why we should not extend our model to include that phase and indeed the next chapter will show mechanisms we could use to ensure the adherence to a workflow definition.

Although the task model is inherently scalable; each task is not aware of where its input data comes from but is aware of where its output goes, the scalability is improved still by the recursive nature of the model. A process has the same structure as a task, allowing processes to be re-used and levels of abstraction applied to the modelling process.

3.7 Runtime Adaptation with Late Instantiated Processes

It is possible to imagine a workflow where the complete structure of the application is not known until runtime. In fact this occurs fairly frequently in the bioinformatics and chemical development fields. In the latter it is common to investigate various chemicals and analyse their suitability for construction of other chemicals. However, based on the results of such analysis different safety considerations must be met later in the development process. What is required is a form of adaptation of the workflow based on the results/experience gained in the early stages (it would be possible, in theory, to enumerate all possible paths the workflow could follow but in practice this is a restrictively large set).

Late instantiated processes are a suitable construct for describing such situations where the entire structure of a workflow is not known beforehand. The contents of a late instantiated processes are not loaded into any enactment environment until it is possible to execute the process in a similar way to lazy loading is used in database systems. It is therefore possible to include a late instantiated process at the end of a workflow and have a task before this that modifies the structure of the process (as it has not been loaded into any environment yet).

Figure 3.23 shows how a workflow might be adapted during execution. The aim of the workflow is to generate and analyse some scientific data. If the analysis is successful the workflow should complete, if the analysis is inconclusive more data must be generated and analysed. The third task in the *analyse* workflow is responsible for configuring the remainder of the workflow. This task could be automated or could be delegated to a human to design the remainder of the workflow. Two potential late instantiated processes are shown depending on whether more data is required. It is also possible that another late instantiated process is used in (b) to recurse and reconfigure the workflow once again. Although this example is very simple it has the advantage that all of the provenance remains with the workflow instance showing how the data was generated and analysed. This is not the case if multiple workflows are executed to further analyse the data: the experiments

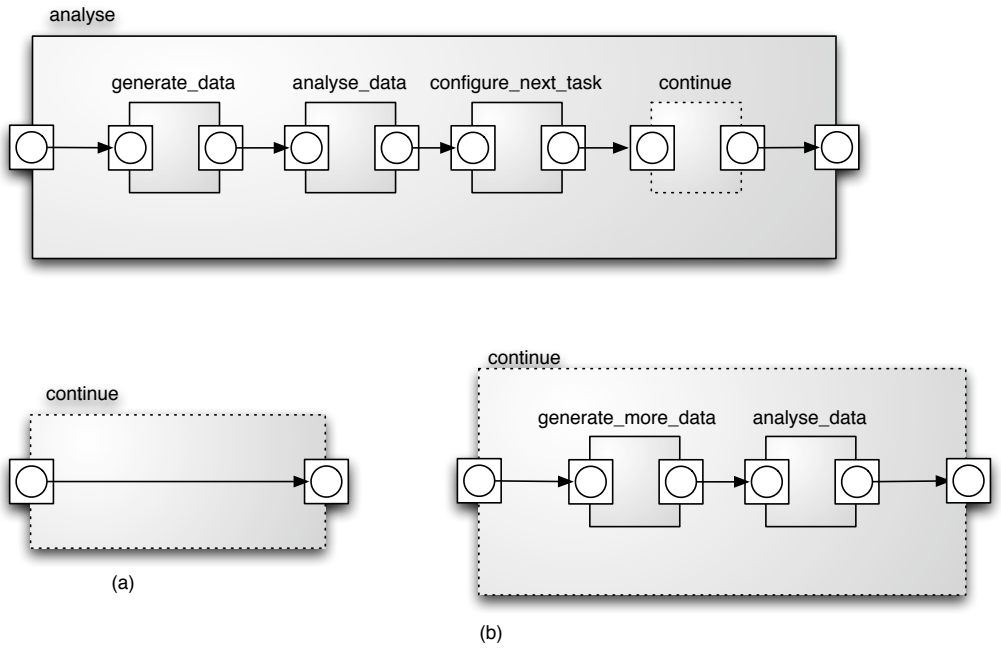


Figure 3.23: Adaptation using Late Instantiated Processes

must be correlated using application specific identifiers etc.

3.8 Concluding Remarks

This chapter has outlined the requirements for workflow specification for Virtual Organisations. As a solution to the requirements we have presented the task model. We have shown its structure, how it is able to represent many of the patterns identified by Van der Aalst and how it maps to a formal model. We have shown how this formal model can be analysed to show that a workflow respects safety and liveness conditions and how application depdendent assertions can be made.

When comparing the Task Model to BPEL the differences become apparent. BPEL is able to model Cancellation patterns and some of the State Based patterns due to its history in petri nets. The Task Model is not able to model these but is able to model patterns such as the Discriminator and N out of M join patterns which BPEL cannot. The most important difference between the two languages is in respect to the amount of state held by a 'task' in each language. The Task Model encapsulates each task with the data needed to execute it, there are no global variables. Conversely, BPEL makes extensive use of global variables to pass data and control the execution. This difference means that whilst it is easy to provide decentralised enactment for the Task Model, this is certainly not the case for BPEL. It is possible to formalise both the Task Model and BPEL but the former can also support custom verifications that describe undesirable application scenarios.

Chapter 4

Service Description

4.1 Introduction

A "service" has become the contemporary abstraction around which modern distributed applications are designed and built. A service represents a piece of functionality that is exposed on the network. The "message" abstraction is used to create interaction patterns or *protocols* to represent the messaging behavior of a service. In the Web services domain, SOAP is the preferred model for encoding, transferring, and processing such messages. Owing to the platform independence of Web Services they are a good choice for performing application integration.

The current technology for describing the interaction patterns of a Web Service is Web Service Description Language, WSDL. WSDL however, does not provide a very rich description of the externally visible behaviour of a service. It defines the structure of the messages that are sent and received by a service but not any description of the relative ordering of these messages (apart from the fact that a response must follow a request). As services become more complex, the number of messages that they may exchange will grow and the interaction patterns of the service will become much more complex. A richer description of the interaction patterns describing the relative ordering of messages helps in a number of ways: we can generate other services (clients for example) which have more knowledge of the service built in. We can verify that one service interacts with another service in the 'correct' manner, as defined in its interface. Finally, we can verify that a set of services interact with each other in the correct manner and show that undesirable situations do not arise, for example that goods are delivered without payment being received.

The SOAP Service Description Language (SSDL) is a SOAP-centric contract description language for Web services [3]. SSDL provides the base concepts on top of which frameworks for describing protocols are built. Such protocol frameworks can capture a range of interaction patterns from simple request-response message exchange patterns to entire multi service workflows within a composite application.

In this chapter, we will introduce the main features of SSDL and its supported protocol frame-

works. The main contribution of this chapter is the presentation of the Sequencing Constraints (SC) SSDL protocol framework for capturing the messaging behavior of Web services acting as part of a composite application or multiparty workflow. The SC SSDL protocol framework can be used to describe multi service, multi message exchange protocols using notations based on the π-calculus. By building on a formal model, we can make assertions about certain properties (e.g. lack of starvation, out-of-order messages, etc.) of workflows involving multiple Web services. We will demonstrate this notion of protocol compatibility and also show the relationship between the Sequencing Constraints and the Task Model presented in the previous chapter. Further, we will show how it is possible to verify that a workflow defined in the Task Model adheres to the protocols defined on the constituent services.

The remainder of this chapter shows how SSDL, and particularly the Sequencing Constraints SSDL protocol framework, achieve the goal of supporting the description of a contract for services. Section 4.2 defines the basic service-oriented model that is espoused by the SOAP processing model. Section 4.3 introduces SSDL contracts and how they can be extended through protocol frameworks. Section 4.4 provides an in-depth look at the Sequencing Constraints (SC) SSDL protocol framework and highlights its formalisation in the π-calculus. Section 4.7 describes how a number of SSDL SC contracts can be checked for compatibility and how a Task Model workflow can be shown to respect sequencing constraints. Finally, conclusions are drawn in Section 4.8.

4.2 Service Orientation

SOAP is the standard message transfer protocol for Web services. However, the default description language for Web services, Web services Description Language (WSDL) [85], does not explicitly target SOAP but instead provides a generic framework for the description of network-exposed software artifacts. WSDL's protocol independence makes describing SOAP message transfers more complex than if SOAP had been assumed from the outset. WSDL's focus on the "interface" abstraction for describing services makes it difficult to escape the object-oriented or remote procedure call mindset and focus on message orientation as the means through which integration is achieved.

The SOAP Service Description Language (SSDL) [91, 95, 3] is an XML-based vocabulary for writing message-oriented contracts for Web services. SSDL focuses on the use of messages combined into protocols (arbitrary message-exchange patterns) to describe a SOAP-based Web service and is intended to provide a natural fit with the SOAP model.[1]

The SOAP processing model [77] in turn provides the fundamental architectural constraints for the Web services stack, as shown in Figure 2.8. While the stack itself is unremarkable, it serves to make the strong point that all Web services must support SOAP and that services interact through

[1] It is assumed that a "Web service" by definition must support SOAP as its native message-transfer protocol.

the transfer of SOAP messages. That is, in a Web services-based environment (which includes workflows composed from Web services) we assume that other communication means (such as Remote Method Invocation (RMI) [58] and Common Object Request Broker Architecture (CORBA) [56]) are merely transport protocols for the transfer of SOAP messages. Such protocols are thus out of scope and do not impact the transfer of messages within the Web services domain.

While Service-Orientated Architecture is not a new architectural paradigm, the advent of Web services has reinvigorated interest in the approach. It is a common misconception that Web services are a form of software magic that automatically corrals an application architect toward a scalable, robust, dependable, and loosely coupled solution. Certainly it is possible to build service-oriented applications using Web services protocols and toolkits to meet such quality-of-service requirements, but, as with any approach and suite of technologies, this is possible only after carefully considering the solution's design and by following the right architectural principles. Furthermore, the use of Web services technologies does not implicitly lead to a service-oriented solution; indeed Web services-based distributed applications could be architected according to the principles of other paradigms, such as resource or object-orientation.

As researchers and developers have rebranded their work to be in vogue with the latest buzzwords, the terms "service" and "service-oriented architecture" (SOA) have become overloaded. In what follows, we treat a service as the logical manifestation of some application logic that is exposed on the network. Such a service may encapsulate and provide access to any number of physical or logical resources (such as databases, programs, devices, humans, etc.). A service's boundaries are explicit, it is autonomous, it exposes message schema information, and its compatibility with other services is determined through metadata information such as policies and protocol description contracts [68]. The interaction between services is facilitated through the explicit exchange of messages. We treat the message abstraction as a first-class citizen of service-oriented architectures and we promote message orientation as the paradigm of choice for enabling the composition of services into workflows.

Services may be hosted on devices of arbitrary size (e.g., workstations, databases, printers, phones, personal digital assistants, etc.), providing different types of functionality to a network application. This promotes the concept of a connected world in which no single device and/or service is isolated. Interesting applications and workflows are built through the composition of services and the exchange of messages.

4.2.1 Messages

A message is the unit of communication between services. Service-oriented systems do not expose abstractions such as classes, objects, methods, or remote procedures. Instead, services bind to messages transferred between them. A number of such message transfers can be logically grouped to form message exchange patterns (e.g., an incoming and a related outgoing message may form a

"request-response"). Such multi message interactions can be grouped to form *protocols* to represent well-defined behaviors.

4.2.2 Protocols, Policies, and Contracts

The messaging behavior of a service in a distributed application is specified by a set of messages and the order in which they are sent and received (i.e., the supported protocols). This is a departure from the traditional object-oriented world where behavioral semantics are associated with types, exposed through methods, and coupled with particular endpoints.

Protocols and other metadata are usually described in contracts to which services must adhere. A contract is a description of the policy that a service supports. As such it will contain the description of the message structure and protocol that a service supports. In addition there will be other functional and non-functional properties specified such as Quality of Service, reliability, access control and security constraints etc.).

4.3 SSDL Overview

The primary goal of an SSDL contract is to provide the mechanisms for service architects to describe the structure of the SOAP messages that a Web service supports. Once the messages of a Web service have been described, any of the currently available (or future) protocol frameworks can be used to combine the messages into protocols that expose the messaging behavior of that Web service. To that end, SSDL defines an extensible mechanism for various protocol frameworks to be used.

SSDL contracts communicate the supported messaging behavior of a Web service in terms of messages and protocols, so that architects and developers can create systems that can meaningfully participate in conversations between them. SSDL contracts may be dynamically discovered (e.g. from registries or equivalent mechanisms) and the protocol descriptions compared against an application's or workflow's requirements in order to determine whether a multi message interaction can sensibly take place.

An SSDL contract is defined in a namespace that uniquely identifies it and consists of four major sections, as shown in Figure 4.1.

4.3.1 Schemas

The "schemas" section is used to define the structure of all the elements that will be used for the description of the SOAP messages. Any schema language may be used to define schema elements, though XML schema is the default choice.

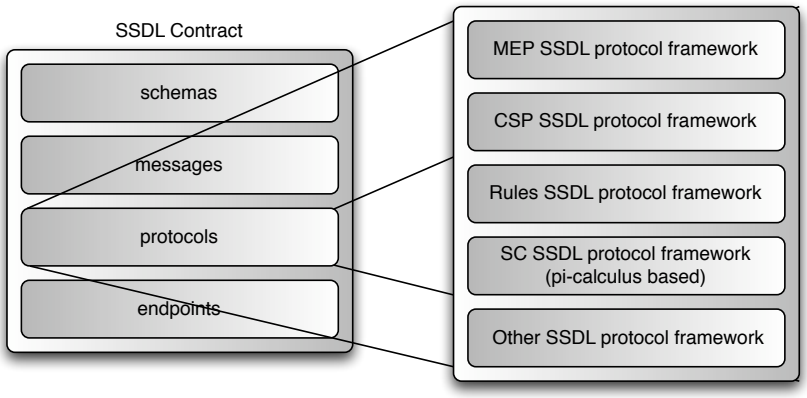


Figure 4.1: The structure of an SSDL contract.

4.3.2 Messages

The "messages" section is where the messages that a Web service supports are declared. There can be many groups of messages defined in different namespaces. However, irrespective of the namespace in which they are defined, the messages included in the SSDL document are all part of the same contract. SOAP messages are described in terms of header and body elements and are named so that protocol frameworks can reference them.

```
<ssdl:messages targetNamespace="uri">
  <ssdl:message name="msg">
    <ssdl:header ref="elements:header1" mustUnderstand="true" />
    <ssdl:header ref="elements:header2" role="urn:ssdl:example:role"/>
    <ssdl:body ref="elements:body1" />
    <ssdl:body ref="elements:body2" />
  </ssdl:message>

  <ssdl:fault name="fault">
    <ssdl:code role="http://www.w3.org/.../role/ultimateReceiver">
      <ssdl:value>Sender</ssdl:value>
    </ssdl:code>
  </ssdl:fault>
</ssdl:messages>
```

Figure 4.2: An example of a message and a fault message

In Figure 4.2, a message `msg` is defined to have two header elements (children of `soap:Header`) and two body (children of `soap:Body`) elements. Note that while the SOAP processing model permits it, the WS-I Basic Profile 1.0a [80] mandates a single element as a child of `soap:Body`. However, SSDL does not enforce that restriction. Figure 4.2 also demonstrates how a SOAP fault message could be declared.

The header element provides the `mustUnderstand`, `role`, and `relay` attributes, which correspond to the equivalent attributes defined by the SOAP processing model (not all of which are shown in Figure 4.2). This makes it possible and straightforward to describe Web services infrastructure protocols.

4.3.3 Protocols and Endpoints

Once the messages in a contract have been defined, we can move on to describe how they may relate to each other. SSDL provides an extensible mechanism based on the concept of protocol frameworks.

A protocol framework uses messages declared in a contract to describe the simple message exchange patterns or multi message interactions that are observed by other services. A protocol framework is an XML-based model for capturing relationships between message exchanges in a workflow and may or may not be supported by an underlying formal model.

It may be possible for the same protocol to be defined in multiple ways using the same or different protocol frameworks. It is up to the designers to choose which protocol framework is best for their needs. Also, it may be possible to translate the description of a service's messaging behavior from one protocol framework to another without losing any semantics, depending on the source and target frameworks.

Some protocol frameworks may be associated with the semantics of a formal model (e.g., CSP, Rules, SC). As a result, it may be possible to use model checkers, such as SPIN [146], Failure Divergence Refinement (FDR) [184] and Mobility Workbench (MWB) [181] to verify the safety (e.g., absence of starvation and agreed termination) and liveness (e.g., eventual termination guarantee) properties of the defined protocols.

The initial release of SSDL comes with four protocol frameworks:

- The *MEP (Message Exchange Pattern) SSDL Protocol Framework* is defined to be a representation of the MEPs defined by the WSDL 2.0 specification [85]. The MEP specification defines the semantics and structure of XML elements representing several message-exchange patterns of two messages at most (excluding faults).

- The *CSP SSDL Protocol Framework* is based on the Communicating Sequential Processes [185] semantics. A protocol is defined in terms of one or more sequential processes that may communicate with each other. Messages that are sent or received represent the events in the described CSP processes [177].

- The *Rules SSDL Protocol Framework* uses preconditions on "send" and "receive" events as the means to describe messaging behaviour. As with the CSP SSDL Protocol Framework, it is possible to use model checkers to verify that a protocol is free from deadlock and race conditions.

- The *SC (Sequencing Constraints) SSDL Protocol Framework* is used to describe multi service interactions, and its semantics are based on the π-calculus [154]. The next section of this chapter discusses this protocol framework in more detail.

An SSDL contract may also define endpoints, such as WS-Addressing Endpoint References (EPRs), of Web services that are known to support the defined contract. While the schemas, messages, and protocols of a contract (the contract is identified by its namespace) remain constant, the endpoints may change. Also, additional endpoints not defined in the contract may exist.

Note that SSDL says nothing about the scope or context of an interaction. A Web service may support one or more instantiations of a protocol at the same time. If more instantiations are supported, a contextualization mechanism is necessary for messages to be associated with a particular instantiation of the protocol (e.g., WS-Context [186], WS-Security [187], WS-Addressing [79] Reference Parameters, service-specific information, etc.).

A detailed description of the SSDL contract and the MEP, CSP, and Rules SSDL Protocol Frameworks are presented by other authors presented in the technical specifications [188, 189, 185, 190].

4.4 The Sequencing Constraints Protocol Framework

The Sequencing Constraints (SC) SSDL Protocol Framework provides a machine-readable description that is used to define the protocols that a Web service supports. Such protocols may be a set of request-response interactions or could use several messages involving multiple parties over arbitrary lengths of time. The framework is intended to provide a simple way of specifying such protocols but also has a formal basis to allow properties of the protocols to be determined if required. Protocols in the framework are specified using a sequential technique, specifying the legal set of actions at each stage of the protocol. It is believed that this leads to a description that is easy to understand, as at each step of the protocol the set of actions allowed is explicitly described. The SC SSDL protocol has a formal basis in the π-calculus, a process algebra for describing mobile communicating processes. The formal basis allows multiple protocols described in the SC framework to be validated to ensure compatibility between them and will be shown in Section 4.7.1

4.4.1 The Structure of an SC contract

4.4.1.1 Describing Actions

The purpose of an SSDL contract is to define the ordering of observable behaviour that a Web Service exhibits. As we observe a Web Service through the messages that it emits and consumes these are the actions we must control. The SC protocol uses the SSDL `msgref` element to refer

to messages that the service sends or receives. There are two mandatory attributes that must be specified when referring to a message: the direction and the participant. The `direction` specifies whether the service is sending the message or is expecting to receive it (out and in respectively). The `participant` specifies which participant in the protocol is the other party in this interaction. At present only unicast communication is supported, although in the future SSDL may be extended to support the specification of broadcast protocols. There is a further discussion of participants in Section 4.4.1.3.

In addition to being able to specify ordering at the granularity of messages being sent and received, SC can also refer to another named protocol. This is done using the `protocolref` element and can be used to provide abstraction and reuse of protocols. For instance, there may be a proprietry protocol to enable authentication with an organisation. This protocol can be defined once and reused whenever the organisation needs to specify that the service requires authentication. In addition it can be used to perform recursion as the protocolref could refer to the current protocol. However, care should be taken to avoid cases of infinite recursion. One way of achieving this is to use a `choice` element to control the recursion which we will see in the following section.

4.4.1.2 Describing Order

In order to describe the order in which the service will send and receive messages we use four constructs: sequence, choice, parallel and multiple. The first three are relatively straight forward and act as expected, the last one, multiple is more complicated. It is possible to nest these ordering constructs to an arbitrary level in order to describe the desired protocol.

Sequence is used to specify that all of the children elements (references to messages, protocols or other ordering constructs) must happen in a sequential order. The previous one (in document order) must have completed (or occurred) before the following one may start.

Choice is used to describe the non-deterministic choice between two or more options. It is non-deterministic in the sense that there is no description of which option should be taken under what circumstances and there are no conditionals to evaluate. It is possible to use the choice element to describe an optional step in the protocol by using a `nothing` element as one of the options. For instance, either send messageA or 'do nothing'. The use of the choice element can lead to race conditions. If we consider the following snippet of SSDL shown in Figure 4.3, we can see that there is a choice element where one option is to send msgA to serviceA and the other option is to receive msgB from serviceA. The problem is that this service may choose to send msgB and the other service decides to send msgA (for simplicity we assume that there is only one other participant and the other SSDL contract is the inverse of this). Equally it is possible that a deadlock occurs if both decide to wait to receive a message rather than sending one. When designing protocols it is generally advisable to avoid situations where there is a choice between sending one message or receiving another [133].

In some cases this cannot be avoided (for instance the WS-Streaming example presented in [95]) and the extra messages must be explicitly dealt with. Such an example is shown in Figure 4.4.

```
<sc:protocol name='race'>
...
<sc:choice>
  <msgref ref="msgA" direction="in" sc:participant="serviceA"/>
  <msgref ref="msgB" direction="out" sc:participant="serviceA"/>
</sc:choice>
...
</sc:protocol>
```

Figure 4.3: Race Conditions using the choice element

```
<sc:protocol name='no-race'>
...
<sc:choice>
  <msgref ref="msgA" direction="in" sc:participant="serviceA"/>
  <sc:sequence>
    <msgref ref="msgB" direction="out" sc:participant="serviceA"/>
    <sc:choice>
      <msgref ref="msgA" direction="in" sc:participant="serviceA"/>
      <sc:nothing/>
    </sc:choice>
  <sc:sequence>
</sc:choice>
...
</sc:protocol>
```

Figure 4.4: Dealing with race conditions using the choice element

To specify that a number of actions may occur in any particular order we use the parallel construct. All the actions must have completed for the parallel construct to be considered complete itself but as the name suggests, they may occur 'in parallel'. In order to specify more complex join conditions the parallel construct can be combined with choice.

The parallel construct is also used when SSDL contracts are not present for all participants in an interaction. It is desirable to be able to specify and verify a contract between n participants where only m of them have SSDL contracts. Situations such as this may occur when third party services that are being utilised and the owning organisations has not provided SSDL descriptions of them. In this case we can take a WSDL description (specifically the message format section) and combine them with the parallel construct. This will mean that the messages can be sent/received an arbitrary number of times in any order. When doing this consideration must be given to whether the WSDL for the service is 'full' (the same problem as described in Section 3.2.5).

The multiple construct has the same semantics as replication in the π-calculus. This means that any children of it will be executed an infinite number of times, in parallel. The use of the multiple

construct is not often required but can be extremely powerful. The most common use is to 'garbage collect' messages when the service is no longer interested in them. For example, the reverse auction pattern often appears in the literature. This pattern involves one party, the buyer, asking many others, the sellers, to quote to provide a service or product. After a certain period of time the buyer chooses one of the quotes and discards the others. Offers may continue to come in after this point but they are discarded. If we are modelling the pattern with SC, the multiple construct can be used to collect these offers that are too late.

4.4.1.3 Describing Participants

One of the factors that distinguishes the SC protocol framework from the other SSDL protocol frameworks is that it is able to support multi-party protocols and protocols where not all of the participants are bound at design time. The participants in the protocol are defined in the contract using the `participant` element and given a unique name. There are a number of ways that a participant can be bound, by which we mean that their endpoint is known. The following rules are evaluated in order until one is found to be true:

1. The first interaction with that party in the protocol is the receipt of a message. In this case the participant who sent the message is bound using data in the message (WS Addressing headers for example)

2. The participant is marked as abstract. A message must be received that is annotated with the `participant-binding-name` that matches this participant before the participant may take part in the protocol. Failure to do this results in a semantic error in the SC contract. The participant is bound using the `participant-binding-content` attribute.

3. The participant is bound by some 'out of bands' method. For instance, the participant may be hard wired into the service or could be looked up in a registry but neither are specified or of interest to the protocol.

It is possible to mask the fact that a participant is bound using one of the messages of the protocol. This is not considered a semantic error in the contract but the contract is not as true a representation of the interaction as possible. In order to maximise the value of the contract (to both the party who exposes it in terms of documentation of the service, and the party who uses it) it is advantageous for the contract to be as close a representation of the observable behaviour as possible.

4.4.2 Example

If we recall the example presented in Chapter 3 consisting of a Buyer, Seller, Finance Organisation and Delivery Company we can discuss their observable behaviour. The SSDL-SC for the Seller is

shown in Listing 4.5.

The protocol that the seller adheres to is the most complex in the scenario as it is the hub of the interaction. The participants in the protocol are shown on lines 6 to 8 and none of them are considered abstract. This means that the Buyer is bound by the receipt of the first message in the protocol (line 12) and the other two participants are bound in an 'out-of-bands' method. In this case it is likely that the finance organisation is fixed as the seller is unlikely to use different banks for different interactions, and the delivery company may be picked dynamically according to business rules. The description of the Delivery service in Listing 4.6 will show an example of an abstract participant.

The protocol that the Seller supports is defined as a sequence of messages, starting with the receipt of the `RequestPriceAvailability` on line 12. This is followed by sending a message back to the Buyer with the price and availability data of the goods they requested. After this the Buyer can choose to do nothing as indicated by the second option of the choice shown on line 42 or can send a `RequestPurchaseOrder` which is received by the other half of the choice mentioned above. This initiates another choice in the behaviour of the seller. Either the purchase order is rejected (perhaps too long has elapsed and the stock is no longer available) on line 21, or, messages are sent to the Finance and Delivery Organisations. The interactions with these two organisations involve sending a request (direcion="out") and receiving a response (direction="in") and occur in parallel. For this reason we use a `sc:sequence` wrapped in a `sc:parallel` element. The `sc:parallel` element is itself wrapped in a `sc:sequence` element as following the interactions with the Delivery and Finance organisations a `PurchaseOrderConfirmation` must be sent back to the Buyer (line 38).

The protocol for the Delivery Organisation shown in Listing 4.6 demonstrates the use of abstract participants which are bound during the protocol. The Delivery Organisation receives a `RequestShippingOrder` message from the Seller and responds with a `ShippingOrder` message. This message could either indicate success or reject from a business perspective but if it is the former then a `ShippingStatus` message will be sent to the Buyer. This message will provide the Buyer with the delivery date of the goods. The reason that the protocol is designed with an abstract participant is that the Buyer is never known to the Delivery Company before the protocol starts. Thus the address to send the notification to is not known until run time. Line 13 shows how the Buyer participant is bound in the protoocl: the `sc:participant-binding-name` attribute indicates it is the Buyer participant that is being bound and the `participant-binding-content` attribute indicates which part of the message structure contains the binding information (this could be any part of the message: either the head or the body). Both of these information items are important as there may be multiple abstract participants bound by the same message.

```
<ssdl:protocols>
 <ssdl:protocol targetNamespace="http://example.org/seller/protocol"
 xmlns:msgs="http://example.org/seller/messages" xmlns:sc=
 "urn:ssdl:v1:protocol:sc">
   <sc:sc>
     <sc:participant name="Buyer"/>
     <sc:participant name="Delivery"/>
     <sc:participant name="Finance"/>
     <sc:protocol name="sellerProtocol">
       <sc:sequence>
         <ssdl:msgref ref="RequestPriceAvailabiltiy" direction="in"
         sc:participant="Buyer"/>
         <ssdl:msgref ref="PriceAvailability" direction="out"
         sc:partipant="Buyer"/>
         <sc:choice>
           <sc:sequence>
             <ssdl:msgref ref="RequestPurchaseOrder" direction="in"
             sc:participant="Buyer"/>
             <sc:choice>
               <ssdl:msgref ref="PurchaseOrderConfirmation" direction="out"
               sc:partipant="Buyer"/>
               <sc:sequence>
                 <sc:parallel>
                   <sc:sequence>
                     <ssdl:msgref ref="RequestFinancingApproval" direction="out"
                     sc:participant="Finance"/>
                     <ssdl:msgref ref="FinancingApproval" direction="in"
                     sc:participant="Finance"/>
                   </sc:sequence>
                   <sc:sequence>
                     <ssdl:msgref ref="RequestShippingOrder" direction="out"
                     sc:participant="Delivery"/>
                     <ssdl:msgref ref="ShippingOrder" direction="in"
                     sc:participant="Delivery"/>
                   </sc:sequence>
                 </sc:parallel>
                 <ssdl:msgref ref="PurchaseOrderConfirmation" direction="out"
                 sc:partipant="Buyer"/>
               </sc:sequence>
             </sc:choice>
           </sc:sequence>
           <sc:nothing/>
         </sc:choice>
       </sc:sequence>
     </sc:protocol>
   </sc:sc>
 </ssdl:protocol>
  </ssdl:protocols>
```

Figure 4.5: SSDL snippet for the Seller Service

```
<ssdl:protocols>
  <ssdl:protocol targetNamespace="http://example.org/delivery/protocol"
    xmlns:msgs="http://example.org/delivery/messages"
    xmlns:sc="urn:ssdl:v1:protocol:sc">
    <sc:sc>
      <sc:participant name="Seller"/>
      <sc:participant name="Buyer" abstract="true"/>
      <sc:protocol name="deliveryProtocol">
        <sc:sequence>
          <ssdl:msgref ref="RequestShippingOrder" direction="in"
            sc:participant="Seller" sc:participant-binding-name="Buyer"
            sc:participant-binding-content=
            "/soap:envelope/soap:header/buyer/wsa:EndpointReference/"/>
          <ssdl:msgref ref="ShippingOrder" direction="out"
            sc:participant="Seller"/>
          <sc:choice>
            <sc:nothing/>
            <ssdl:msgref ref="ShippingStatus" direction="out" sc:partipant="Buyer"/>
          </sc:choice>
        </sc:sequence>
      </sc:protocol>
    </sc:sc>
  </ssdl:protocol>
</ssdl:protocols>
```

Figure 4.6: SSDL snippet for the Delivery Service

4.4.3 The relationship between SC and workflow

Protocols within SSDL are intended to describe the externally visible behaviour of services through message exchanges. Workflow technologies as described in Chapter 3 describe the internal structure of a service in terms of tasks which send and receive messages. There is a clear relationship between the two technologies. In fact the relationship is threefold:

1. **Verifying that a workflow respects a SSDL SC Contract**

   It is possible to verify that a workflow 'respects' the SC contracts of the services that are used within the workflow. By 'respects' we mean that the structure of the workflow always sends and receives messages that are legally allowed by the services that are being interacted with. For instance, the workflow that enacts the Seller service in our example sends and receives messages that are legal in the Buyer, Delivery and Finance Services.

   This verification is described in more detail in Section 4.7 but has many uses. A primary one is when an organisation wants to describe consistency constraints on their internal services using SSDL. It is possible to verify that all the workflows that utilise those services to model the business processes do not break the consistency constraints.

2. **Generating Workflow stubs from SC**

'Contract First Development' is a term that is frequently used to describe the process of developing a distributed system by first developing the contract that each service (or component) will expose [191]. If we consider SSDL SC as a suitable contract for contract first development it appears natural to develop these first, verify that the contracts are compatible with each other and then generate services which adhere to these contracts. It is possible to reduce the load on the developers of the services, and reduce the likelihood of errors by generating workflow stubs from the SSDL SC protoocol. This method is not dissimilar from the endpoint projection used in WS-CDL[21].

The problem with generating workflow stubs from SSDL SC is that that there are many implementations of the workflow which will adhere to the SSDL contract. This means that the stubs generated may be very different from the one required to implement the business logic of the service. However, sometimes the stubs will be close to the required implementation. In addition it is possible (and necessary) to verify that the resulting service still respects the SSDL SC contract in the manner described in point 1.

3. **Generating SC from Workflow**

   It is also possible to perform the converse of point 2 and generate the SSDL SC contract from the workflow. This involves taking the workflow and computing the externally visible behaviour. In essence, all of the internal communications are removed, leaving only the communications with the remote services. The internal communications which themselves are controlling the order of the tasks are replaced with the explicit ordering from SSDL SC. This is described further in Section 4.7.2.

4.4.4 SSDL and Service Interaction Patterns

In Chapter 2 we mentioned the work that has gone into identifying common interaction patterns amongst services. We noted that there are criteria used to group the patterns into coherent groups, namely the number of parties involved, the number of messages involved and whether the recipient of a message is the same as the initiator of the response. Specifically, the 13 patterns are shown in Table 4.1 and along side we show whether SSDL-SC is able to directly represent the pattern (+) or not (-). If it is possible to partially model the pattern or use a work around we show +/-.

As we can see from Table 4.1 many of the Service Interaction Patterns are directly supported. Some have primitives in SSDL that correspond to the Service Interaction Pattern such as send or received. For others it is trivial to combine a number of send/receive actions to support the pattern. It is more interesting to focus on those patterns that are not supported by SSDL-SC or only partially supported.

The only pattern that is not supported at all is number 10, atomic multicast notification. This

Number	Pattern Name	SSDL-SC
1.	Send	+
2.	Receive	+
3.	Send/Receive	+
4.	Racing Incoming Messages	+
5.	One to Many Send	+
6.	One from Many Receives	+/-
7.	One to Many Send/Receive	+/-
8.	Multi-Responses	+/-
9.	Contigent requests	+/-
10.	Atomic Multicast Notification	-
11.	Request with referral	+
12.	Relayed Request Pattern	+
13.	Dynamic Routing	+/-

Table 4.1: Representing Service Interaction Patterns in SSDL

pattern is defined as the ability of a party to send a number of messages to other parties and a number of the parties are required to accept the notification within a certain timeframe. There are two problems with supporting this pattern in SSDL-SC. Firstly, there is no notion of atomic multicast in SSDL-SC, only unicast. In the future we may investigate the addition of other transmission primitives such as multicast but these will have other effects on the formalisation. The second problem with supporting this pattern is that SSDL-SC has no notion of time. It is not possible to specify timing properties between messages, only the order of those messages relative to each other. As this pattern requires the responses to be received within a certain timeframe we cannot describe it. In reality this pattern is only realisable by using a channel that supports transactional activities. Indeed the authors of [99] mention this fact when they are describing a potential solution to the pattern.

The reason that patterns 6-9 and 13 are not directly supported is also because of the lack of timing support. Pattern 6, One from Many receives specifies that a number of messages must be received from different parties and correlated together. In order to do the correlation they must arrive in a timely manner and success of the interaction may depend on the number of messages received. Clearly we cannot describe what is meant by 'timely manner' but there is a version of the pattern that does not take time into account. In the un-timed version the number of messages received is fixed but the time limit is unbounded. We are able to represent this using SSDL-SC but as we cannot represent the timed version we only show a +/- in the table. Patterns 7,8 and 9 have similar constraints and work-arounds.

4.5 Breaking Encapsulation Using SC

It is normal practice that a service will encapsulate a number of resources and business logic. It is not apparent to the 'users' of that service what resources are being encapsulated. This is generally considered to be advantageous as it allows the owners of the service to change the implementation without affecting the users (as long as the interface remains the same). Indeed it is becoming common to provide 'value added services' where a number of existing services are enhanced or combined to provide a new service. Often very little new logic is necessary but the existing services have been leveraged in a new manner. A simple example is a service that provides analysis of data (perhaps gene comparisons). The code that does the analysis is provided by one organisation but the data is provided by another.

It is common in fault tolerance fields to use redundancy to provide a higher level of service availability. Therefore someone wishing to utilise the data analysis service might design their business logic to fail over to another analysis service if the first one failed. However, if the second service also utilises the same database then there is a single point of failure that the user is unaware of. The user is not achieving a higher degree of fault tolerance and the extra effort required to handle the fail over is wasted. Such a scenario is shown in Figure 4.7.

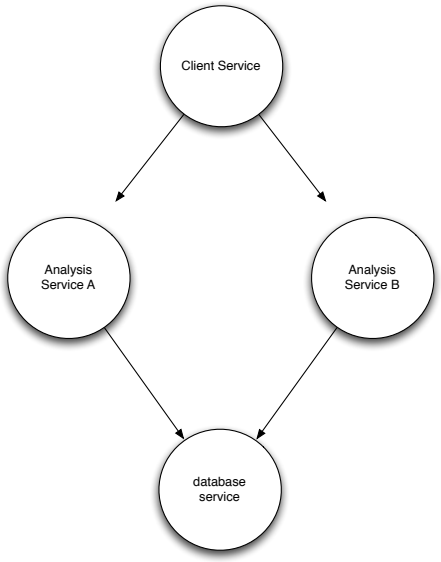


Figure 4.7: Breaking encapsulation using SSDL

SSDL SC provides a solution to this problem, although it is up to the service providers to dictate when it is appropriate to use it; there are times when encapsulation is necessary and breaking it would be harmful. SSDL SC allows the providers of the analysis service described above to explicitly state that another participant will be providing the data. This would be achieved by having another participant in the protocol and showing the messages that the service will send and receive with

the data service. If both of the analysis services provided this it would allow the users to make an informed decision about the levels of fault tolerance provided by a fail over based solution.

4.5.1 Limitations of Sequencing Constraints

It is not feasible to develop a generic protocol framework that can easily represent all protocols, and any attempt at such is likely to result in a framework that is very cumbersome to use. The four initial protocol frameworks complement each other, in many cases what is difficult to model in one framework is easy in another. In the Sequencing Constraints framework it is not possible to represent protocols that have an unknown number of participants. All of the participants in a SC protocol must be known at design time even if they are not bound. The example of a reverse auction mentioned earlier is impossible to model in the SC framework in a generic manner as the number of Sellers is not known at design time. It is possible to create a number of protocols that deal with a certain number of Sellers but this is not a a realistic work around. In future work we may explore the possibility of extending the SC framework to be able to deal with protocols that have an unknown number of participants - the introduction of participant roles and a multicast primitive might be a starting point for the investigation.

The SC protocol framework is not very good at representing protocols that have messages which have no ordering constraints on them. For instance when a service can receive a `cancel` message at any time. In SC this must be explicitly stated as a choice between receiving a cancel message and receiving a different message. In fact the problem is more severe: If the protocol is at a step where it is due to send a message care must be taken to avoid race conditions as described in Section 4.4.1.2.

4.6 Formalising SC

4.6.1 π-calculus representation of SSDL SC

In order to ascertain whether the protocols defined in SSDL Sequencing Constraints satisfy certain properties we can translate them into a π-calculus representation. The properties that we are interested in are principally whether or not a number of protocols are compatible with each other. We use the word 'compatible' to describe a combination of safety and liveness properties. For instance, we would like to be able to show that Service A will never send a message the Service B cannot receive at that time (a safety condition). Also, we would like to know that eventually Service A will send a certain message that B is expecting to receive. This liveness condition is sometimes referred to as a lack of 'starvation'.

In order to be able to verify the compatibility of protocols it is necessary to translate them into a π-calculus form. We are fortunate that the XML representation shown earlier is structurally

very similar to the π-calculus representation that we require. In fact this was one of the design decisions made whilst formulating the SC protocol framework. In the π-calculus representation of the Sequencing Constraints, each protocol is represented as a π-calculus process and these processes are connected via channels, one per process. These channels can be thought of as the endpoint that the participant listens to. References to other protocols are represented using parametric calls to other π-calculus processes and the msgref element maps onto either sending or receiving a name along the channel to the other participant.

Within each SSDL SC protocol the ordering elements are translated into their π-calculus equivalents. The simplest element, sequence, simply maps onto the π-calculus ".". Thus service a sending msg1 to service b and then receiving msg2 would become $\overline{b} < msg1 > .a(msg2)$. The choice element in SSDL SC is represented using the π-calculus non-deterministic choice operator "+". When required we can test for name equality (when a name is received) using the $[x = y]a(x).P'$ notation. This means that the process will only continue as P' if the name y is recived along the channel a. This is used when there are more than one receive action in the choice options to distinguish between the different messages that might be received (and potentially along different channels).

The parallel element in SSDL SC is slightly more complex to handle than the two that we have described above. Naturally we use the π-calculus "|" operator that represents parallel composition but we need to do a little more work in order to be able to 'join' the processes once they have finished executing. We represent each parallel activity as a process that executes in parallel with the main 'thread' of the protocol. We also introduce some private channels for the processes to communicate with each other. There is one channel per parallel process that allows the main thread to indicate that the parallel thread should start. Along this channel a name is sent that is used to indicate when the thread has completed. The same completion name is given to each of the parallel threads that are in that 'group' (each of the activities inside that parallel SC element). The completion name is used to count the number of parallel processes that have completed. When the same number that have been started have completed the main thread of the protocol can continue. This is illustrated in the π-calculus for the Seller service shown below:

$$Seller(seller, buyer, delivery, finance) = (\nu\ d, f) \tag{4.1}$$

$$(seller(rpa).\overline{buyer} < pa > . \tag{4.2}$$

$$(seller(rpo).\overline{d} < r > .\overline{f} < r > .r.r.\overline{buyer} < poc > \tag{4.3}$$

$$+\ 0)) \tag{4.4}$$

$$\mid d(comp).\overline{delivery} < buyer, rso > .[x = so]seller(x).\overline{comp} \tag{4.5}$$

$$\mid f(comp).\overline{finance} < rfa > .[x = fa]seller(x).\overline{comp} \tag{4.6}$$

$$\tag{4.7}$$

The multiple element in SSDL SC is directly translated into the replication operator of the π-calculus. Since there are an infinite number of replicas of the sub-elements of the multiple element there is no need to try and join the threads when they have finished executing, indeed it would be impossible (or at least not terminate). The multiple element is rarely used in protocol descriptions but is included here for completeness.

The nothing element has no direct translation onto the π-calculus but it is often represented using the "0" termination operator. The nothing element is most commonly used as one option in a choice element indicating that the choices are optional. Assuming that no elements exist following the choice element the "0" operator can be used to indicate that explicit termination is one of the choices.

In our discussion of the formalisation so far we have paid little attention to the partners involved and their mobility. This is perhaps the most important aspect of the SC protocol framework and that which distinguishes it from the others: the ability to represent the dynamic nature of an interaction. To express the dynamism we use the inbuilt channel passing mechanisms of the π-calculus.

If we recall the SC description of the protocol that the delivery company follows shown in 4.6. It specifies that there are two other participants, the seller and the buyer. The latter is marked as `abstract=true` which means that it is unbound initially. Therefore the delivery company is unable to send a message to the buyer until the participant is bound. This starting architecture is shown in Figure 4.8. The protocol for the delivery company specifies that the Buyer participant is bound by the contents of the RequestShippingOrder (line 13 of Listing 4.6. This binding is effectively the passing of a π-calculus channel from the Seller to the Delivery Company, shown in Figure 4.9 and results in a new system structure, shown in Figure 4.10.

When considering the formalising of the protocol we must be able to introduce this channel passing into the desrciption. The problem that arises is that in the XML representation the binding

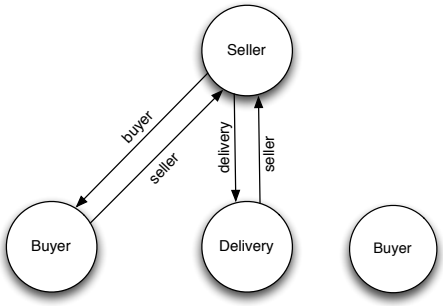
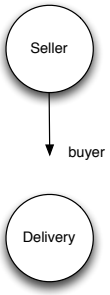


Figure 4.8: Initial System

Figure 4.9: Channel Passing



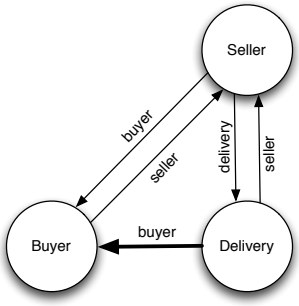


Figure 4.10: Final System

information is only present on the receiving participant whereas in the π-calculus representation we need this information at both the sender and the receiver. This mismatch stems from a design decision taken on the SC protocol framework. The XML representation was created as a pragmatic way of describing protocols. It was created with the principles of message orientation in mind [192], one of which is that the sending party has no control over what the receiver does with the message. Thus it is not possible to make assumptions/stipulations such as 'the receiver must bind a participant using this message'. However, it is perfectly legitimate for the receiver of a message to explicitly state 'I am binding a participant with the contents of this message' as once they have received the message it is in their 'jurisdiction'. Thus the XML representation does not have the notion of sending a channel to another party but the other party does have a notion of receiving a channel.

To perform the translation into the π-calculus representation we use a polyadic version of the π-calculus. The messages that bind participants are turned into n-name receives where n is equal to one more than the number of participants being bound by that message (so it includes the data itself). An example of this is shown in the π-calculus for the Delivery Service below. The buyer participant is bound by the receipt of the `rso` message from the seller.

$$Delivery(delivery, seller) = \tag{4.8}$$

$$delivery(buyer, rso).\overline{seller} < so > .\overline{buyer} < ss > \tag{4.9}$$

The second half of the translation can only be done at the time when the compatibility verification of a number of protocols occurs. This step involves identifying the message that is being sent to bind the participant and add the channels that need to be sent. It is a semantic error if the sending participant does not have a participant with the same name as the one being bound by the receiver of the message. An example of this (the other half of the Seller/Buyer/Delivery example is shown

on line 4.5 of the π-calculus representation of the Seller's protocol above.

4.7 Verification of SSDL SC

There are two aspects concerning the verification of protocols described in the Sequencing Constraints notation: firstly that n services are compatible with each other; and secondly that a workflow written in the Task Model notation respects the protocols of the services that it uses. We should remember that the notion of compatibility that we are concerned with is the lack of 'starvation' (eventually a service receives a message it requires) and the lack of message being sent that cannot be received at that time.

4.7.1 Contract Compatibility

In order to verify that a number of contracts are compatible with each other it is necessary to combine them into a single 'system'. This system contains the contracts for the individual participants combined in parallel with appropriately scoped channels to allow communication between the different processes. If we consider the example given above the system will include the Buyer, Seller, Delivery and Finance contracts combined using the parallel operator '|'. The π-calculus processes for the Buyer and Finance Organisation followed by the complete system (named Interaction) are shown below (The Delivery and Seller processes were shown earlier):

$$Buyer(buyer, seller) = \overline{seller} < rpa > .buyer(pa). \tag{4.10}$$

$$(\overline{seller} < rpo > .buyer(po). \tag{4.11}$$

$$(buyer(ss) + 0) \tag{4.12}$$

$$+ 0) \tag{4.13}$$

$$Finance(finance, seller) = \tag{4.14}$$

$$finance(rfa).\overline{seller} < fa > \tag{4.15}$$

$$Interaction = (\nu\ buyer, seller, delivery, finance) \tag{4.16}$$

$$Buyer(buyer, seller) \mid Seller(seller, buyer, delivery, finance) \mid \tag{4.17}$$

$$Delivery(delivery, seller) \mid Finance(finance, seller) \tag{4.18}$$

We can see that the Interaction introduces new names for *buyer, seller, delivery, finance* and these names are used as channels to communicate between the processes. In fact the snippets of the definitions provided in the previous sections are incomplete without these definitions. Following from that the processes representing each contract are simply combined in parallel.

In order to test for compatibility we use the same method as outlined in Chapter 3, namely weak open bi-simulation. However, before we can perform bi-simulation analysis it is necessary to consider what we mean by compatible. There are a number of definitions possible and it may be a domain specific problem. Here we will decide that contracts are compatible if at the end of the interaction there are no services waiting to send or receive any messages. This indicates two things: firstly, a lack of 'starvation' as no services are waiting to receive any messages. Secondly it shows that there are no messages that are sent but cannot be received as there are no messages waiting to be sent.

In order to show that the contracts are compatible we will try and show that the set of contracts are weak open bi-similar to the null process, 0. If we recall our initial discussion of weak open bi-simulation we can see that it ignores internal actions (τ) and reduces the processes using the reduction rules. This includes pairs of prefixes that perform complimentary actions (send/receive) as they form a *regex*. Thus, if the contracts are compatible they will reduce completely reduce until there are no interactions remaining resulting in the null process 0. Therefore if the contract interaction is weak open bi-similar to the 0 process the contracts are compatible.

In a similar vein to compatibility it is possible to show that the contracts when combined together display other properties too. For instance, we can augment the contracts (during verification) to show that the goods are not delivered if the payment fails (under this set of contracts it is not possible but as mentioned in Chapter 3 we could imagine a set where it is possible. To achieve this we would introduce an action that we know does not have a complimentary action. For instance we could introduce $\overline{paymentFail}$ and $\overline{deliverGoods}$ actions at the relevant places. If we then test whether the resultant process is bisimilar to $\overline{paymentFail}.\overline{deliverGoods}$ we will find whether the two can occur in sequence. If the processes are bi-similar we know that this situation is possible, if not it is not.

4.7.2 Workflow Verification

Chapter 3 presented a workflow language for describing the executable business processes of a Virtual Organisation. Although SSDL and indeed the Sequencing Constraints framework are agnostic of the implementation details of any of the services being described it is interesting to investigate the links between the two technologies. The difference between the Task Model and SSDL is similar to that between concrete and abstract BPEL: SSDL describes the observable behaviour of the service and the Task Model describes how this behaviour is realised (i.e. the implementation). The two compliment each other and separate the concerns between those which are implementation specific that those that are not [193].

An obvious desire is to be able to assert that a workflow described in the Task Model adheres to the SSDL specification of its behaviour. This will allow us to partition the verification of a system into a number of stages:

1. Check that the workflow is structurally sound as described in Section 3.5.1
2. Check that the workflow adheres to the SSDL-SC description of its observable behaviour
3. Check that the set of SSDL-SC contracts that comprise the system are compatible as described in the previous Section.

Unfortunately there is an impedance mismatch between the Task Model and SSDL-SC: more information is contained within the latter than the former. To overcome this we must add the extra information into the Task Model. Some of the information is implicitly contained within the Task Model but must be made explicit and some information is not captured and so must be added. We also require a slight change to the model itself. We will present these in order and then move onto an example showing the verification of the workflow for the Delivery Company defined earlier.

The first item that we must address is that the Task Model does not contain all the information about different participants in the workflow. The Task Model treats each task as equal and does not correlate tasks that interact with the same service. This means that although we know that message B was sent after message A we do now know whether they were sent to the same service. Clearly we need this information for the SSDL description so that we can send the messages along the correct channel. Some of the information relating to service correlation does exist in the Task Model but only for the tasks which send messages, not those that receive messages. For the messages that are being sent we know which endpoint they are being sent to. Therefore we can assume that one endpoint corresponds to one service and therefore one SSDL description (this assumption only amounts to syntactic sugar and can be overridden if we know that endpoints do not correspond one-to-one with services). However, the Task Model does not contain any information about the where messages are being received from. It might be the case that there is only one other service

involved in the workflow but we do not know this in the generic case. Therefore the first extra information we must add to the Task Model is an attribute on each task to define the service being interacted with. This must be unique for each logical service within the workflow but is used for correlation purposes only - the text used does not mean anything.

The second change that we must make to the Task Model concerns participant binding within SSDL. This information is not captured in the Task Model and the Task Model actually assumes that all parties are known at design time. One of the key features of SSDL-SC is that it is able to describe protocols where the participants are now known until during the protocol execution. We will extend the Task Model slightly to allow workflows to be defined where the participants become instantiated during the execution. In Section 3.2.8 we discussed data dependencies and stated that the source and sink of a dependency is a part of a message defined in WSDL. In order to deal with dynamic binding of participants it is necessary to relax this statement. In addition to WSDL parts being the source and sink of dependencies we will allow other properties such as message headers and task properties to be sources and sinks of dependencies. The property that we are particularly interested in is the endpoint of the service being invoked which is a property of a task. This allows us to have one task which receives a message containing the address of another service and to route this information via a data dependency to the endpoint property of another task. In the case of the Delivery Company's workflow (shown in Figure 4.11, the RSA task will receive both the shipping order and the endpoint of the Buyer. The input set of the SS task will contain an additional part which corresponds to the endpoint the Shipping Status will be sent to. Then we can connect these two parts with a data dependency.

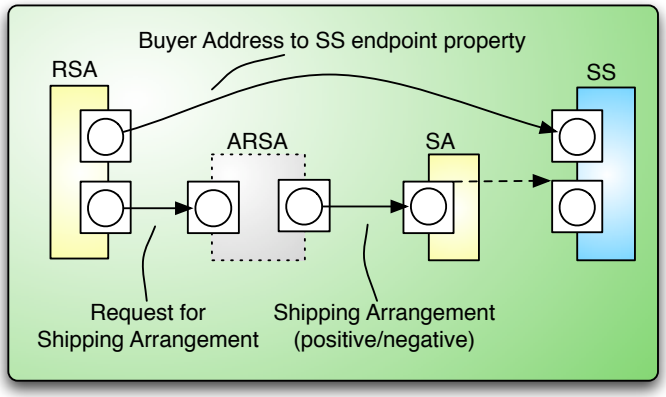


Figure 4.11: Extended workflow for the Delivery Service

Once we have made these alterations to the Task Model we are able to verify that a workflow adheres to an SSDL-SC contract. Again we use weak open bi-simulation but this time we will determine whether the SSDL-SC contract is bi-similar to the workflow described in the Task Model.

We can use the π-calculus process for the delivery company that we defined earlier but we must modify the π-calculus process for the Delivery company's workflow to include sending and receiving the messages from other parties. This involves replacing the $\tau_{interact}$ placeholder that was shown in Section 3.4.2 with send and receive actions. We use the service attribute that we mentioned above to know which service the message is being sent to (eg. Buyer, Seller) and include one name for each part that is in the data set (input set, output set). For instance, the $\tau_{interact}$ in the RequestShippingArrangement is replaced with $delivery(buyer, rso)$ to indicate that we are receiving a message over the delivery channel that contains two names: one corresponding to the buyer's channel and one corresponding to the request for shipping order. The $\tau_{interact}$ in the ShippingArrangement task is replaced with $\overline{seller} < so >$ where we are sending the shipping order to the seller.

With these changes it is possible to use the π-calculus processes defined in Chapter 3 and compare them to the SSDL-SC π-calculus processes defined earlier using weak open bi-simulation. If the processes are similar the workflow adheres to the SSDL-SC contract, if not it does not. We should also note that there may be some tasks which communicate with services that we do not want to include in the SSDL-SC contract. For example, the Analyse Request for Shipping Order task invokes a service internal to the organisation to process the request. We do not want to (in this example) check the SSDL-SC corresponding to that service. Therefore we can leave the $\tau_{interact}$ in place and the interaction will be ignored. If we wanted to include it we could have combined the internal service's SSDL and the external SSDL for the Delivery Company in parallel and this would have checked both contracts at the same time.

4.8 Conclusions

SSDL is a contract language for describing message-oriented, asynchronous interactions between Web services. In addition to its simplicity and SOAP-centricity approach, SSDL is also able to capture rich conversations between Web services without being limited to simple request-response message-exchange patterns as is the case with WSDL.

A novel and powerful aspect of SSDL is that it enables the use of protocol description frameworks that are amenable to formal verification. While this is certainly a luxury for today's simple Web services systems, as the size and number of connected services in a deployment increases, the ability to formally verify that the system as a whole, or individual services, will not starve or race is an extremely useful proposition.

These contracts can be encoded in the Sequencing Constraints protocol framework that supports dynamic multi party contracts. They are amenable to formal reasoning with respect to their compatibility and we can exploit their relationship with workflow. Finally we have demonstrated how it is possible to verify the compatibility of both SSDL SC contracts and workflows represented in the

Task Model Notation.

Chapter 5

Workflow Enactment

5.1 Introduction

A programming system would not be complete without a method of enacting those programs. As the Task Model is aimed at providing a programming system for Web Services, specifically those that model business processes in Virtual Organisations, the provision of an engine to enact the workflows is a necessity. The contribution of this Chapter is the presentation of the architecture for a distributed enactment engine and a realisation of this architecture in our prototype workflow engine, DECS. When designing the Task Model notation a great deal of care was taken to ensure that the language did not unnecessarily restrict the architecture that could be used to enact it. We saw in Chapter 2 that Virtual Organisations may take many different architectural forms and there is no 'one size fits all' solution. The same can be said for architectures of enacting workflows, flexibility is required to allow each Virtual Organisation to choose the style that suits their requirements. We will further discuss the requirement for flexibility and suggest some common distribution patterns in Section 5.2.1.1.

The DECS workflow engine presented in this Chapter has been designed and implemented using component middleware, namely Java 5 EE (Enterprise Edition) [160] and specifically making use of the Enterprise JavaBeans 3 (EJB) aspect of the specification [194]. EJB components require an application server to execute in and DECS has been targeted at the popular Open Source application server, JBoss [161]. The current version has been tested in version 4.2.2.GA of JBoss. There is no reason, in theory, that the application could not use other application servers that implement the JEE specification. However, as is described in the following sections, certain facilities of JBoss are utilised and the use of a different application server would require modification to some components.

The remainder of this Chapter is structured as follows: Initially we will discuss the requirements for a distributed workflow system and then present a high level architecture that meets these requirements. One of the key requirements we will focus on will be the need for flexibility of enactment. Following on we will describe the implementation of DECS, our prototype distributed workflow

engine and provide an example of the way it executes. Finally we will draw our conclusions.

5.2 Requirements for workflow enactment in Virtual Organisations

The enactment of workflows in any application domain requires scheduling tasks to run based on the rules which are defined in the workflow specification [170]. A task is an application specific *unit of work* which in this case we will take to correspond to the invocation of a web service. Rules define when a task may be executed based on the state of the current execution. The scheduling of tasks therefore relies on the evaluation of the current state, the determination of what tasks may begin execution and the updating of the state based on the tasks that have completed execution.

The execution of a workflow may take anything from seconds to months to complete. Within this time it is possible that machines may fail, services may become unavailable, transient network failures may occur and other undesirable and unforeseen circumstances manifest themselves. It is necessary to minimise the effects of these problems through the use of fault-tolerance techniques and in many cases mask them to the users of the system.

As an organisation deploys an increasing number workflows, performance degradation should not be experienced. A scalable architecture is required to implement this where there is no reliance on a single central entity. If achieved, this would allow an organisation to deploy and run an arbitrary number of workflows concurrently.

Perhaps the area where enactment of workflows in Virtual Organisations differs from in other arenas is the requirement for flexibility. Virtual Organisations encompass multiple physical organisations, each wishing to retain a certain degree of autonomy. Each physical organisation may be a member of a number of Virtual Organisations and have different policies for each with respect to data access, service availability and so on. In addition to this, the question of 'who owns the shared workflow in a Virtual Organisation?' can be raised. Some Virtual Organisations may employ a Trusted Third Party to enact the shared workflow, others may trust one member (often the primary member) to enact the workflow. In other scenarios, distributed enactment can be utilised to allow each organisation to enact parts of the shared workflow. As this requirement for flexibility is so relevant to Virtual Organisations we will investigate potential distribution patterns in the following section.

5.2.1 Centralised Enactment

Workflows may be coordinated centrally, as shown in Figure 5.1. This is the simplest scenario, where a central server (WFE stands for Workflow Engine) sends and receives all of the messages

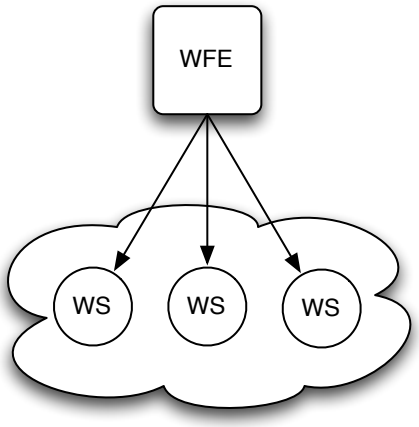


Figure 5.1: Centralised Enactment

necessary to complete the workflow. Centralised enactment is sometimes likened to a hub and spoke formation. The enactment engine is situated at the hub of a wheel and the services that are being invoked are situated at the rim of the wheel at the end of the spokes. All of the messages flow along the spokes, to and from the enactment engine. Centralised enactment is simple and offers advantages for speed of development and debugging. Depending on the size and characteristics of the workflow, centralised enactment might be the most efficient form as there are no coordination messages that need to be sent. However, we will examine workflows in the following sections which are not suited to centralised enactment.

There are two examples of where centralised enactment is likely to be used: firstly when the location (both geographical and organisational) of the services being used is either unknown or not important and secondly when a Trusted Third Party (TTP) is used to enact the workflow on behalf of a Virtual Organisation. The workflow shown in Figure 5.1 fits into the first of these categories. The workflow engine is shown interacting with a number of Web Services 'in the cloud'. The location of these services is not important to the designer of the workflow. It might be the case that the services reside within the organisation enacting the workflow if, for example, the workflow is performing application integration or for a Virtual Organisation whose implementing the Star or Market Alliance.

The workflow shown in Figure 5.2 shows a similar architecture but includes organisational boundaries. Here although centralised enactment is used, some of the services reside within one organisation and some reside in another. This Figure shows one possible configuration of a Virtual Organisation where Organisation A has the responsibility of enacting the workflow that uses some of each organisation's services. Another possible architecture for a Virtual Organisation is shown in Figure 5.3 where a TTP is being employed to enact the workflow on behalf of the two organisations. The

services that the workflow utilises are accessible from outside the organisations (accessible by at least the TTP).

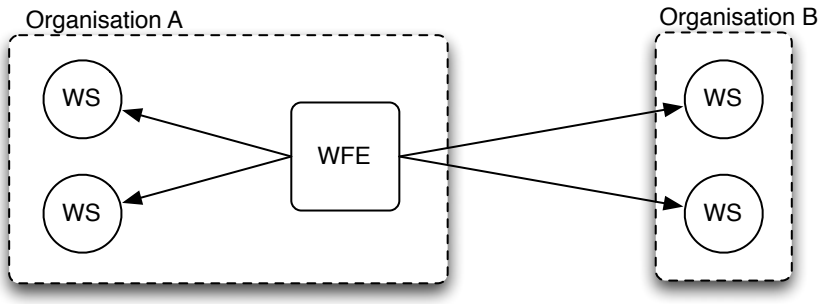


Figure 5.2: Centralised Enactment between organisations

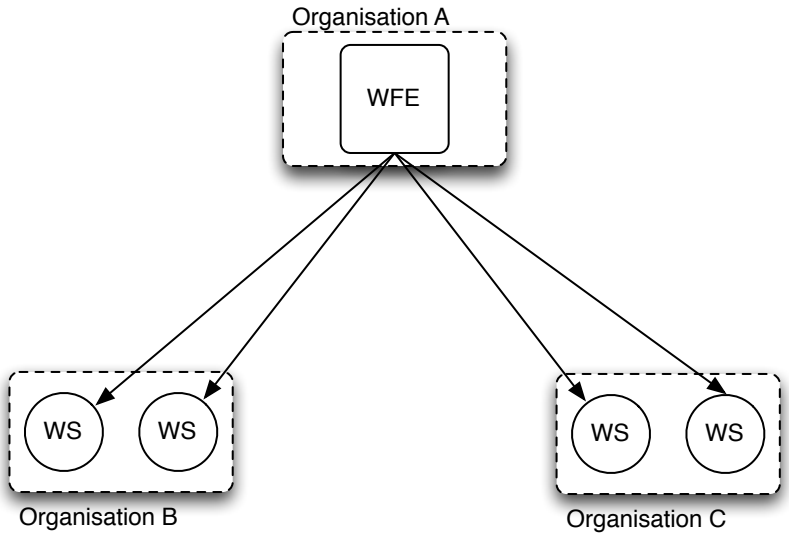


Figure 5.3: Enactment by a Trusted Third Party

5.2.1.1 Distributed Enactment

The opposite of centralised enactment is decentralised, or distributed, enactment where two or more enactment engines coordinate to execute the workflow. Let us consider some of the situations where distributed enactment is advantagous:

To reduce Data Transfer A common pattern in both scientific and business reporting workflows is to extract data from a database and then analyse it to derive knowledge. It is possible that the application performing the analysis is geographically close to the database containing the data. However, using centralised enactment, the data must be transferred to the workflow

engine, stored locally and then transferred back to the analysis application (as shown in Figure 5.1). If the workflow engine is not close to the database and analysis application, and when considering very large data sets this is not a desirable situation. A more efficient architecture is to decentralise the enactment and locate a workflow engine near the database and analysis application (or preferably co-located). The fragment of the workflow which deals with data extraction and analysis may then be deployed within this workflow engine. Thus, the data must only be transferred from the database to the local workflow engine and then onto the analysis application. Once the analysis has completed, the results can be sent to their intended destination (as specified in the workflow fragment). This has increased the design time effort but has saved the potentially costly and time consuming long distance transfer of the data sent to and from the workflow engine. The architecture for such a scenario is presented in Figure 5.4 and an example in Section 5.4.10.

Increase Parallelisation It is not uncommon in bio-informatics/genomics environments to have workflows which have tens or hundreds of tasks which may execute in parallel. For instance, when a new gene is sequenced, scientists wish to re-run previous analysis to gauge its significance. Such tasks are usually isolated and may execute in parallel, however a centralised enactor will not be able to achieve the maximum level of concurrency possible. By partitioning the workflow across multiple enactment engines, a greater degree of parallelism can be achieved. It is possible to automatically analyse and partition workflows to achieve greater concurrency using well known algorithms such as Sarkar's algorithm for code partitioning [195].

Organisational Requirements It is possible that within a Virtual Organisation a workflow may be partitioned for organisational reasons related to information security. For instance, a workflow could be partitioned along organisational boundaries (e.g. Figure 5.4 or 5.5. The latter presents a less strict partitioning). If some of the services which are being composed in the workflow are internal to an organisation, acting on sensitive data for example, it is likely the organisation would not wish to allow access to the service from outside their firewall. However, a large inter-organisational workflow could be designed as a single, or hierarchical, entity and then partitioned so that each organisation enacts part of it. This limits the flow of sensitive data to within the organisation and only the data which each partner organisation explicitly needs must cross the organisational boundary. Another similar reason for partitioning the workflow along organisational boundaries can be considered as the response to the question "Who owns the shared workflow in a Virtual Organisation?". In such organisations, there is no single owner or natural enactor of the workflow. It might be possible to enlist a TTP to enact the workflow (Figure 5.3) but this too could be undesirable. In this situation it is natural to partition the workflow and allow each organisation to enact the part that interacts with their

services.

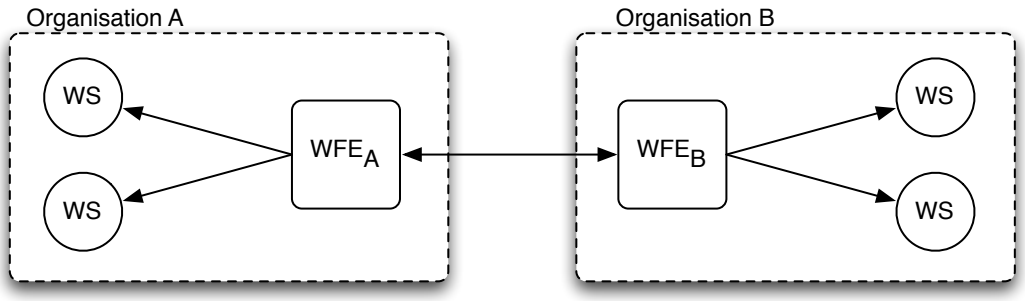


Figure 5.4: Distributed Enactment with services only available internally

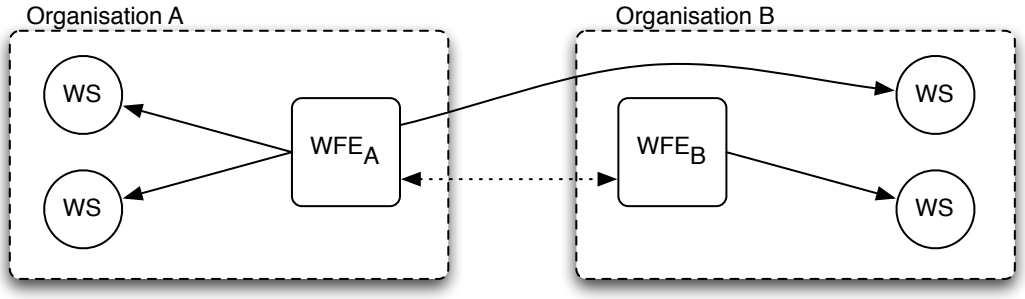


Figure 5.5: Distributed Enactment with some services available externally

5.3 Architecture of a decentralised workflow engine

Before considering the implementation of our decentralised workflow engine we must discuss the abstract architecture of such a system[196]. We will avoid implementation and platform specific details, instead focusing on the the components that are likely to be present in any implementation. It is worth noting that this is one possible design and other ones which are equally elegant may exist.

The first component to consider is the control loop. The control loops is responsible for analysing the current state of the system and deciding if any actions may be taken. In this case, actions will be the invocation of a service somewhere either local or remote. The control loop must update the current state when actions happen and then perform the analysis again in a looping fashion. It is possible that the control loop is constantly executing or that it is reactive to a change in state. The latter avoids the busy/waiting problem by waking the component when there is a possibility that forwards action can be taken [197]. Tightly coupled with the control loop is the data store. This is

used to persist the data associated with each definition of a workflow and each instance of a running workflow. It is the data for each instance of a running workflow which comprises the state of that workflow and is analysed by the control loop. The data store could be a database (relational, object oriented or other), could be in memory or could be any other form of data storage such as flat files on a filesystem. It is likely that for fault tolerance reasons the data will be written to disk periodically as determined by the application.

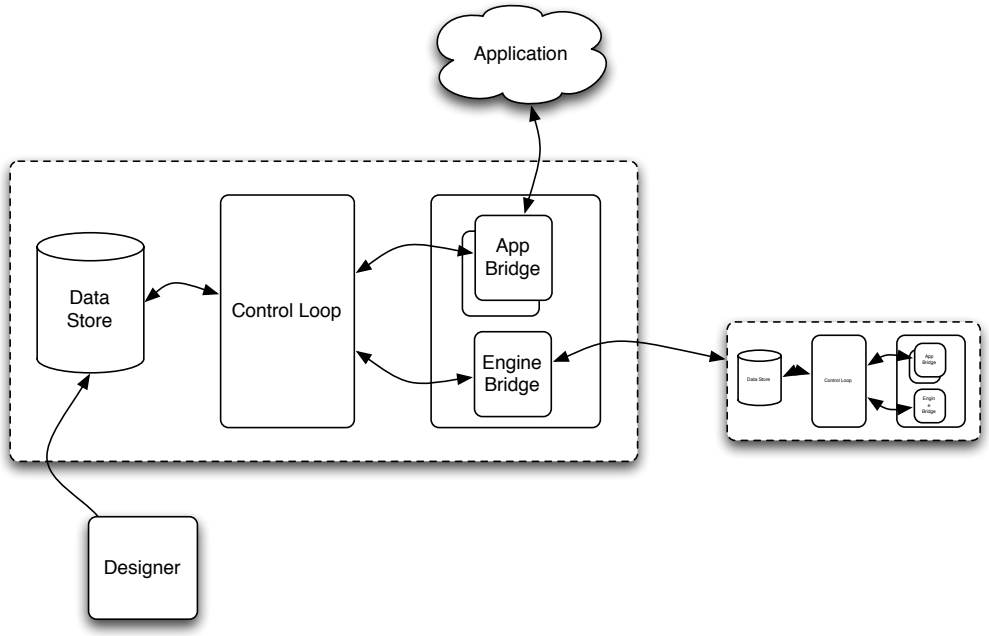


Figure 5.6: Abstract Architecture of a distributed enactment engine

The data for each definition of a workflow must be designed using a component of some sort, either graphical or textual. This must then be saved in the data store to allow the control loop to instantiate it when required. The designer component may be online or offline but we consider it to be outside of the core 'engine'.

When considering the execution of the workflow, there are two aspects that must be taken into account. Firstly methods of performing the functionality of the workflow and secondly, methods of interacting with other workflow engines. The former we will call an application bridge as it acts as a bridge between the control loop and the applications that the workflow invokes. The application bridge is responsible for invoking and interacting with applications which are outside the workflow engine, for example, databases, Web Services, shell scripts, Java applications etc. These applications which are invoked by the application bridge perform the actual 'work' of the workflow. There are likely to be a number of application bridges which interact with specific technologies. However, they should share a common interface to allow the control loop to interact with each of them in a common

manner. It should be noted that there might not be a single common interface but multiple ones depending on such things as the interaction style. For example there might be one for synchronous interactions and one for asynchronous ones. The application bridge is also responsible for interacting with applications that can be considered clients of the workflow. Varying technologies might be used to initiate the workflow and the application bridge in question will interact with the control loop to begin a new instance of the workflow.

The Engine Bridge is a specialised case of the Application Bridge where more is known about the application being interacted with. The application bridge simply knows the technology being used to interact and any non-functional attributes such as transactional and security requirements. The engine bridge knows that it is interacting with another workflow engine and therefore can use a predefined configuration and may in some cases optimise the interaction. For example a secure channel may be used and workflow instance identifiers will be propagated across the interaction. In addition it may be possible to 'bundle' multiple messages into one larger one to make an efficiency gain.

5.4 Design and Implementation of DECS

This section will describe the implementation details of DECS, a decentralised peer-to-peer based enactment engine that realises the architecture described in the previous section. The high level design of DECS is shown in Figure 5.7. The following subsections will firstly describe an example of how a workflow is enacted and then proceed to describe the purpose and internal structure of each of the components in turn. Using the Java EE environment allows DECS to make use of the rich set of functionality which Java EE application servers provide. For example, uniform access to persistent storage, flexible transaction control and a unified security model. DECS utilises the Entity Beans, Session Beans, Message Driven Beans and JMS from the Java EE architecture.

5.4.1 Example

This example will introduce the components shown in Figure 5.7 and show how they interact with each other when enacting a workflow. We will consider the example shown graphically in Figure 5.8 which has one process, P which contains two tasks T1 and T2 which should execute sequentially with T2 following T1. There are three dependencies in the example: one from the processes input to the input of T1 (a), one from the output of T1 to the input of T2 and one from the output of T2 to the output of P. We will assume that the process has been deployed into DECS successfully and an endpoint has been exposed to allow it's invocation.

Initially a client (not shown) will send a SOAP message that will be routed to the ProcessInitiator and part of the context associated with this request will contain the XML description of the workflow.

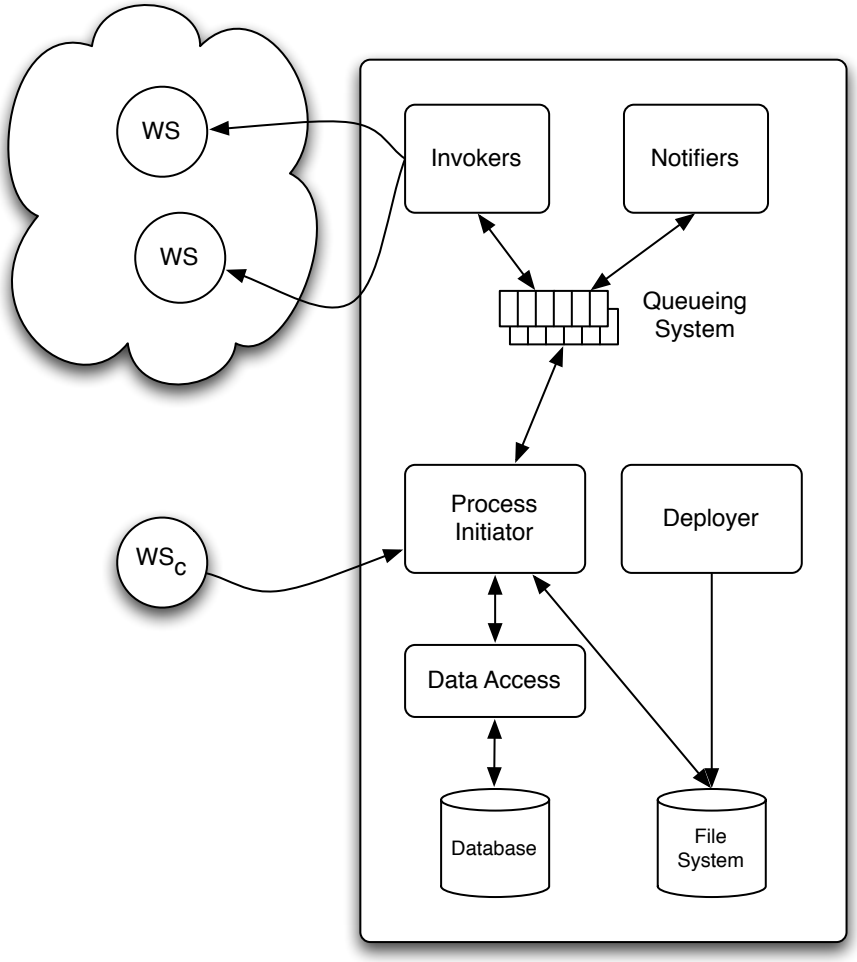


Figure 5.7: DECS System Architecture

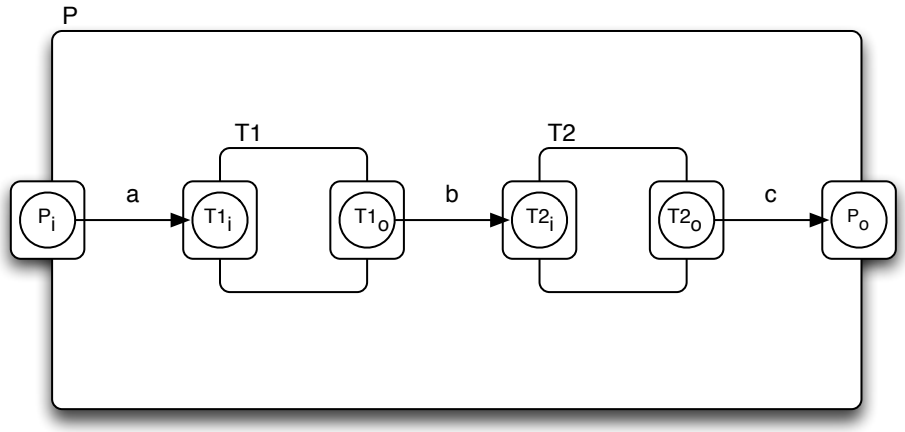


Figure 5.8: Simple Workflow Example

The SOAP message received by the ProcessInitiator corresponds to the Process's input P_i. The ProcessInitiator will create and persist the artifacts for the workflow such as objects to hold the dependencies, task data etc and then place a message on the NotificationQueue. This message will cause the DependencyNotifier to be invoked which will analyse all the the dependencies whose source is the P_i, in this case only a. The data that 'flows' along a will be persisted and a message will be put on the InvocationQueue indicating that a has been fulfilled. In this example the dependency a is really a group that only has one member but we will talk about dependencies rather than groups for readability. The TaskInvoker will retrieve the message from the InvocationQueue and evaluate T1 to see whether it can be executed (T1 is the sink of dependency a). As there is only one dependency and this has been fulfilled T1 can be invoked. Therefore the SOAP message will be built. This will be constructed from the data from dependency a with an XSLT transform applied to it and will be persisted as $T1_i$. The message will be sent to the remote endpoint and the TaskInvoker will block waiting for a response. This response will correlate to $T1_o$ and once it has been persisted a message will be placed on the NotificationQueue indicating that task T1 completed with message $T1_o$. In this example this is the only possible completion 'state' but others were shown in the example in Section 3.5.

The message on the NotificationQueue will in turn force the DependencyNotifier to execute which will evaluate and persist dependency b and indirectly invoke the TaskInvoker again via the InvocationQueue. The TaskInvoker will construct and send the SOAP message to T2 and then place a message representing the output $T2_o$ on the NotificationQueue. The DependencyNotifier will evaluate $T2_o$ and persist the data for dependency c and then inform the TaskInvoker that c has been fulfilled. On this invocation of the TaskInvoker the execution will be different to previous times. As the fulfilment of c means that the output of the Process, P_o is complete a message will be put on the ResultsQueue. This message will be consumed by the ProcessInitiator and contains the payload of the response message sent to the client. The ProcessInitiator persists this data and sends this message which corresponds to the completion of the workflow.

5.4.2 Deployer

The Deployer is the component that is most tied to the JBoss application server as it makes use of the internal APIs exposed for deploying applications. The following discussion is centred on the use of those APIs and if another application server was to be used the deployment component would need to be re-written accordingly.

The purpose of the Deployer is to take a workflow described in the Task Model XML notation and expose it as a Web Service. Once this has been done, other Web Services are able to invoke and interact with it. We will call those Web Services 'clients' although we are not enforcing a traditional client-server relationship. There are two ways of 'invoking' the deployer: firstly, it is exposed as a

Web Service itself. This means it is possible to send a SOAP message to it where the payload of the message is a Process Definition (i.e. Task Model workflow).

Secondly, the JBoss server contains a special directory which is continually monitored. This is known as the 'deploy' directory and JBoss will attempt to deploy any file which is placed in it. The suffix and extension of the filename dictate how JBoss will deploy the file. The DECS Deployer registers the suffix `-process.xml` with the JBoss MainDeployer [198] component. This results in the JBoss MainDeployer delegating the deployment of any file with that suffix to the DECS Deployer.

The implementation of the Web Service method of deployment is trivial: it simply validates the payload of the message received and writes it into a file in the 'deploy' directory. This file is then picked up by the MainDeployer and delegated to the DECS Deployer. The Deployer Web Service is has both 'synchronous' and 'asynchronous' operations to perform the deployment of a workflow. If the asynchronous version is invoked WS-Addressing headers must be present to indicate where to send the response to. The synchronous version keeps the channel alive and sends the response back along this channel. The response the 'client' will receive contains the success of the deployment and a GUID (Globally Unique Identifier) that can be used to route messages to this workflow (for instance to remove the definition). If the workflow contains any sub-deployments the top level workflow will only be deployed if all of the sub-deployments are successful too. If any of the deployments fail all the others are rolled back to leave the system in a consistent state.

The Deployer is invoked by the MainDeployer to process files which have the `-process.xml` suffix. It is passed a DeploymentInfo object as a parameter which is an encapsulation of the complete state of the component, in this case a workflow, being deployed. The Deployer is able to retrieve the raw XML description of the workflow and convert it into Java objects that are more convenient to work with. The conversion into Java objects is done using an implementation of the Java Architecture for XML Binding (JAXB). As part of the build process for the DECS engine, Java objects are created from the XML Schema that describes the Task Model notation using JAXB tools. The resulting objects are essentially data holders and accessors for all of the elements which can be present in a workflow description. It is possible to automatically 'unmarshal' an XML file into these objects and then manipulate them in the Deployer component.

The first task that the Deployer performs, having populated the Java objects from the XML is to look for any SubProcesses that are present in the description. If any are found they are marshalled into an in-memory description and passed to the MainDeployer. This will have the effect of a recursive call to the DECS Deployer. The reason for deploying any SubProcesses before the main Process Definition is that we want to be able to fail the main deployment if any of the SubProcess deployments fail.

The act of actually deploying the workflow has a number of steps. The first step is to dynamically

create a Java Web ARchive (WAR file) 'on the fly'. [1] This is created in one of the JBoss temporary directories and has the same structure as a traditional WAR file [199]. A number of files are placed into the WAR file, including the XML Process Definition and other XML files that JBoss requires to expose a Web Service (configuration and data mapping files). The DECS engine is packaged with defaults for each of these files but they must be configured before they are added to the WAR file. The configuration replaces default names with those given in the Process Definition for things like the name of the service, the URL that it will be accessible at, the target namespace of the service, etc. As most of these files are XML, XPATH is used to select the desired place to make the change and then the element/attribute is altered. The final modification that must be made is to add the Types information to the WSDL file that describes the interface to the process being deployed. An alternative to performing this change would be to use an `xsd:anyType` in the WSDL description but this would not be very useful to the clients of the service or any tools that processed the WSDL. Further documentation would be required to specify the format of the messages that the service accepted/emitted. Once the WAR file is has been constructed, the location of it is passed to the JBoss Main Deployer. The MainDeployer will detect this WAR file and deploy it in the same way that it would deploy any other Web Application

The way that the WAR file (and thus the Web Service) is written means that all invocations of the workflow will be handled by one 'generic' service, implemented by the Process Initiator component (described in the following Section).

In order to be able to enact distributed workflows, the manner of the distribution must be described. This is achieved by annotating each task in the workflow with the address of the machine hosting the instance of DECS which will enact that task. The XML description of the Task Model notation contains an attribute which can be placed on the task element to allow this: the `host` attribute. The host attribute can be placed on any task or process except the top-level process. If the host attribute is not present it is assumed that the DECS node which initiated the deployment will enact that task. The DECS node that initiated the deployment will also enact the top level process and will expose an endpoint which allows the workflow to be invoked.

If the workflow is utilising distributed enactment, the other nodes that are responsible for enacting part of the workflow must also deploy it. The difference between the other nodes and the main node is that only the main node will expose a Web Service interface to interact with the service. Therefore, the deployment on the other nodes is a little simpler. SOAP messages containing the workflow definition are sent to the other nodes. As no interface need be exposed they do not need to dynamically build a WAR file as described above, but can simply store the file and associate it with the GUID that identifies that workflow.

[1] The reason for deploying services in this manner is that JBoss is able to expose 'traditional' Web Services in this way.

It should be noted that the Deployer component is also responsible for the 'un-deployment' of the workflow. There is very little to do in this case, simply remove the WAR file and ensure that any components further up the deployment hierarchy also undeploy the aspects that they are responsible for. A mapping is kept between the GUID that was generated in the Deployer for each workflow and the name of the WAR file that implements the workflow. This mapping is interogated and the relevant file removed, resulting in the removal of the Web Service interface and the definition of the workflow. If the workflow has distributed components, SOAP messages are sent to the other servers that coordinate this workflow and they will remove the relevant files too.

5.4.3 Process Initiator

The ProcessInitiator component is responsible for dealing with all invocations of workflows. When a message is sent from a client, regardless of which workflow (i.e. Web Service) it is sent to, the message is routed to the ProcessInitiator. When the ProcessInitiator is invoked it has a context associated with it which is specific to the Web Service Endpoint that the message was sent to.[2] This is similar to any other Java web application which all have contexts such as the ServletContext if the web application is implemented as a Servlet. It is this context that allows the generic ProcessInitiator to specialise for each workflow implementation. Specifically, the item in the context that we are most interested in is the XML description of the workflow that has been invoked.

When invoked, the ProcessInitiator is principally concerned with two things: the SOAP message that was sent to invoke it and the XML workflow description that it must enact. Both of these XML artifacts are parsed and Java objects representing them are created using JAXB in the same way as in the Deployer component. The ProcessInitiator inspects these objects and creates Java Entity Beans to hold this data in the database. In theory it would be possible to store the JAXB objects in the database directly. However, this has a number of problems: the APIs to the objects, although adequate, are not very convenient to program; it is easier to abstract the data into objects that have a good API. Secondly the objects created by JAXB are not in a format that creates an efficient database design: they are not in a normal form as they are not intended to be used as the foundation of a database schema.

Once the objects representing the workflow (tasks, subprocesses and dependencies) have been created and persisted to the database, a message is put on a JMS queue to inform the Notifiers that there are dependencies that can be fulfilled. The structure and principles of the DependencyNotifier will be described in Section 5.4.5.

Workflows can be invoked in two manners: request-response or two one-way invocations. These styles are often referred to as synchronously and asynchronously but in distributed systems the words synchronous and asynchronous have more precise meanings which do not necessarily hold

[2]The context is actually instantiated during the initialisation of this instance of the ProcessInitiator.

here. When a workflow is invoked in a request-response manner the client of the workflow keeps the communication channel open (often HTTP) for the duration of the workflow. The results of the workflow are sent back to the client along this communication channel and it is then closed.

When a workflow is invoked in a one-way manner a message with an empty payload is returned to the client. The headers of this message contain a GUID that uniquely identifies this instance of the workflow. This GUID can be used in the future by services to route messages to this instance of the workflow. The response to the workflow, if one exists, is sent along a new communication channel from the server to the client in a 'call-back' type interaction. The address that the response should be sent to is included in the original request using WS-Addressing headers. This address is usually the location of the client but not necessarily; it could be an arbitrary third party. In the Task Model notation the response part of a request-response interaction is modelled as the output message(s) of a Process in the workflow. A callback invocation is identical to any other task within the workflow.

If the workflow is invoked in a request-response manner the ProcessInitiator is keeping the communication channel open to send the response back to the client. In this case, once the workflow has been started by putting a message on the NotificationQueue, the ProcessInitiator will wait for a message on the ResultsQueue (with the correct process identifier). When this message is received it's payload is inserted into the body of a SOAP message which is returned to the client. In request-response mode any problems that occur are propagated to the client via a SOAP fault describing the error. Such problems include internal exceptions in DECS and any unexpected exceptions from the Web Services being invoked.

If the workflow is invoked in a one-way manner the ProcessInitiator will complete after putting the message on the NotificationQueue. As described above, the response will be sent to the client in the same way as any other Web Service request by the TaskInvoker. Should any problems occur during the processing of the workflow, they are propagated to the client by sending a message to the fault address listed in the WS-Addressing headers if this exists. If the fault Endpoint Reference does not exist in the headers the failure is masked by DECS and the workflow attempts to continue executing.

If the workflow that is being invoked contains tasks which are being coordinated by other DECS instances, i.e. distributed enactment, these instances must be notified of the workflow invocation too. This is done by sending a SOAP message to the other nodes which contains the GUID of the workflow that has been invoked and also the GUID of this instance of the workflow. This allows the remote node to instantiate the Java objects that they are responsible for with the workflow and ensures that the different DECS instances all have the same identifier for this workflow instance.

The ProcessInitiator of each DECS instance will instantiate local objects for the enactment of the workflow under certain circumstances:

Tasks If the task (or process) is being enacted by that instance of DECS then task objects will be created. If the task is being enacted by a different instance of DECS then the task is ignored and the remainder of the workflow description is processed.

Dependencies If the source or sink task of the dependency is being enacted by this DECS instance then a local object for the dependency is created. This is because each instance only needs to 'know' about the data that either it will generate or it will consume.

5.4.4 Invokers

In order to understand the Invokers and Notifiers it helps to look a little more at the messaging infrastructure within DECS and the relationship between these components. In Figure 5.7 these components were linked with the ProcessInitiator by way of a message queueing system. Figure 5.9 shows this aspect of DECS in more detail. The diagram shows that the ProcessInitiator, Invokers and Notifiers are connected by three queues. These are implemented using JMS topics which have one-to-many semantics but we will use the word 'queue' to refer to them. The important properties of the queues is that they are persistent and transactional. This means that users of the queues deposit and retrieve messages inside a transaction ensuring consistency in failure scenarios and that the state of the system can be rebuilt following a failure.

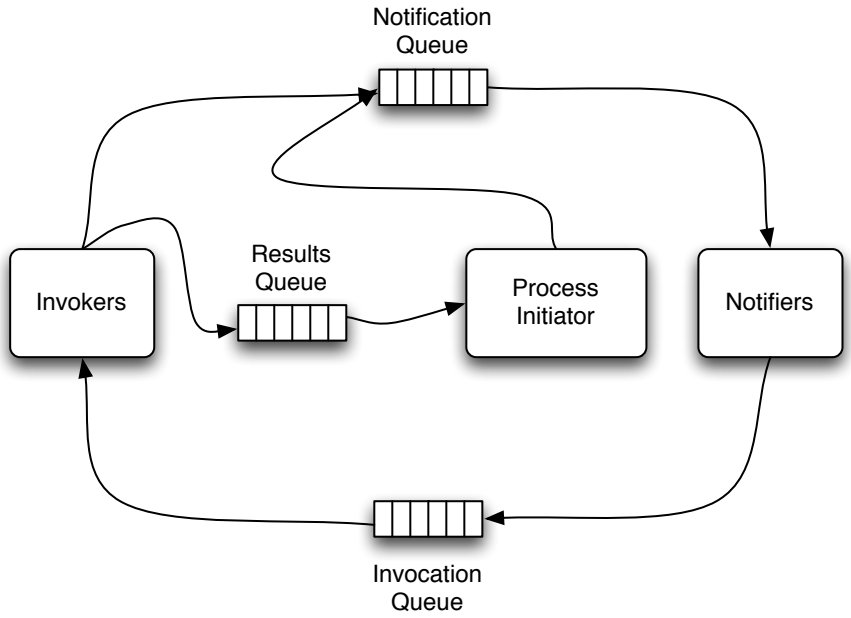


Figure 5.9: Messaging System Architecture

The Invokers are responsible for invoking tasks that are invocations of Web Services, i.e. sending SOAP messages. At present only invocations of Web Services can be described in the Task Model

notation but the Invoker architecture has been designed to be extensible. It would be straightforward to implement invokers that interacted with other technologies such as Java RMI, HTTP/REST based services, Java function calls etc. The TaskInvoker, an instance of an Invoker, is implemented as a Message Driven Bean [200]. A Message Driven Bean (MDB) is linked to a JMS Queue or Topic and is called when a message is deposited on that Queue or Topic. In this case the TaskInvoker is invoked when a message is put onto the InvocationQueue and this message is passed as a parameter to the onMessage method.

The onMessage method of the TaskInvoker deals with two cases: firstly, a message must be sent to a Web Service invoking a task; secondly a response must be sent to a client which indicates the 'result' of the workflow if the workflow is invoked in a synchronous manner. Each time that a dependency is fulfilled within the workflow the TaskInvoker will be invoked. The TaskInvoker must determine whether the act of fulfilling this dependency has meant that another task can execute. If so, the TaskInvoker should build and send the SOAP message to the Web Service but if the task has other dependencies which are not fulfilled then the TaskInvoker will do nothing.

If we recall the semantics of dependencies in the Task Model (from Chapter 3) we see that there is a mixture of 'AND' and 'OR' semantics. Dependencies can be grouped in order to introduce alternatives and data redundancy. If more than one dependency is fulfilled from a particular group then the one with the highest priority is used. At least one dependency from each dependency group must be fulfilled in order for the task to be executed. The TaskInvoker checks this AND/OR relationship and determines when a task can be executed. When this is so, the SOAP message to be sent to the Web Service is constructed. The construction is a two stage process: firstly an in memory XML document is formed and then that document is transformed using XSLT to create the body of the SOAP message. The use of XSLT to produce the message has two advantages: firstly, it is possible to perform simple data transformations when the data formats do not exactly match; secondly it means that in the future it will be easier to support other formats than SOAP. The SOAP message is then sent to the remote Web Service and if a response is expected the communication channel is left open. The details of this response are persisted and a message is sent to the Dependency Notifier via the NotificationQueue.

When the TaskInvoker determines that a response must be sent to the client as the workflow has completed execution (assuming it was invoked in request-response manner) it must pass the response message to the ProcessInitiator to send. This is because the ProcessInitiator is holding the open communication channel to the client. The TaskInvoker builds the SOAP message using the two stage process detailed above and puts it on the ResultsQueue. This is retrieved by the ProcessInitiator who returns it to the client.

5.4.5 Notifiers

The Notifiers and specifically the DependencyNotifier is a simple component that is responsible for evaluating the dependencies within a workflow. It is implemented as a MDB that responds to messages on the NotificationQueue. Messages are placed on this queue when a task completes execution and therefore the dependencies whose source is that task may be able to be fulfilled. In order to fulfil the dependencies we must first find the ones whose source is the correct task and message. A dependency has two 'ends', a source and a sink which are messages that belong to a task (In Chapter 3 we referred to these messages as the input and output 'set' of a task). The majority of the time the source is an output message of a task and the sink is the input message of another task. The data associated with a task can be considered to 'flow' along the dependency from source to sink. Given that a task may complete with one of a number of possible messages (e.g. normal, named faults) only those dependencies on the specific message will be fulfilled. Once the set of dependencies has been found the data that they contain is persisted. The act of fulfilling a dependency might mean that the task at it's sink is now executable. To communicate this fact a message is sent to the TaskInvoker telling it to check if that task is now executable.

The description above has assumed that tasks are the sources and sinks of dependencies. There are two special cases that must be considered: when the workflow starts, the 'outer' process is the source of the dependencies and when the workflow completes the 'outer' process is the sink of the dependencies. These cases are actually dealt with in exactly the same manner described above, the dependencies are found, the data is persisted and the TaskInvoker checks any task or process at the sink of the dependency to see whether it's state can change. In this way, the ProcessInitiator places a message on the NotificationQueue to indicate the start of the workflow and the Notifier checks all dependencies on the input of the workflow, the tasks are executed and so on.

So far we have only considered the case where the two tasks that are connected by a dependency are being coordinated by the same instance of DECS. In distributed enactment this is not necessarily the case. If the sink of a dependency is a task that is being hosted on a different DECS node a SOAP message is sent to that node containing the details of the dependency that has been fulfilled. For instance, the identifier of the dependency, the GUID of the workflow instance being executed and the data that is flowing along the dependency. The remote node will persist this data and then send a message to the TaskInvoker to see if the task is now invokable in the same way as with non-distributed dependencies.

5.4.6 Data Persistence

The creation and storage of data relating to workflow execution is an important aspect of DECS. Some of this data is required in order to execute the workflow and some is required to provide

provenance of each workflow execution [172]. The primary method of data storage and retrieval in DECS is using the Java Persistence API (JPA) which provides mechanisms for persisting Java objects to a database. DECS makes use of the MySQL database with InnoDB tables for persistent storage [201]. InnoDB tables are not the default for MySQL as their performance characteristics are less favourable than MyISAM tables. However, InnoDB tables offer transactional capabilities which are not available in the other tables and are required in this implementation.

Transactional (ACID) semantics are required in a number of places within the DECS architecture and if transactional semantics are lacking undesirable situations may occur. The most visible situation will occur with the interaction between the DependencyNotifier and the TaskInvoker. If the access to the data is not serialised it is possible that tasks may not execute (depending on race conditions). This is because it is possible for the the TaskInvoker to check to see if a task is executable shortly before the DependencyNotifier completes writing the data that makes the task executable. In this situation there will not be another JMS message on the InvocationQueue and so the TaskInvoker will not check that task again, resulting in a stalled execution. It would be possible to devise a different architecture for DECS that did not rely on transactional semantics for data access but this is beyond the scope of this thesis.

As described above, the JPA is used to persist the data from within DECS. There are three Java objects that hold the data: one for Processes, one for Tasks and one for Dependencies. Each of these has member variables which are mapped onto database fields using Object-to-Relational mapping techniques. These objects are created at the beginning of the execution of the workflow and populated with data as the workflow continues to execute.

Many scenarios such presented in the bioinformatics, chemistry and medical fields require 're-producability' of results. This means that results published should be reproducible by a different scientist using similar techniques and the same data set. To facilitate this as much 'provenance' from each execution of a workflow as possible is stored in the database. For example, the time and date when the workflow was executed, the time that each task took to execute, the exact messages that were sent to each Web Service and the responses received, the state of each dependency etc. The aim of storing this data is to allow the workflow execution to be reconstructed at a later date. It is possible to imagine a tool that would show this graphically and allow the user to 'fast-forward' or 're-wind' a workflow, viewing its state at any given point.

5.4.7 Error Handling

DECS treats errors in workflows in different ways depending on the type of error. If the Web Service being used within a workflow lists SOAP faults as part of its interface in the WSDL definition, the workflow designer can use these within the workflow definition. It is possible to define different actions dependant on which fault is received from the Web Service. For example if a payment service

returned a fault indicating that the payment was rejected it is possible to attempt to re-invoke it with different parameters (a different credit card acquired from the user for example). However, if the service returned a different fault a different course of action could be taken. We call such SOAP faults listed in the WSDL as expected faults. They are known at the time the workflow is designed and it is possible to attempt forward error recover [197] following them. The other class of faults are known as unexpected faults, which are not known at design time and not described in the interface to the service. Often these faults are returned to the user of the workflow (either via the return channel or the fault-to WS-Addressing header), wrapped in a message trying to give them as much information as possible. The user is then able to diagnose the problem, fix the workflow and re-invoke it. The HTTP status code [202] of these faults is often in the 500 range indicating an internal server error or 401 indicating that authorisation has failed. There are a few unexpected errors that DECS is able to recover from. For instance, errors showing that the service has moved (301, 303) allows us to redirect the service invocation to the new location. These errors are masked and the user is unaware of them.

5.4.8 Scalability

The JBoss application server can be run in two different modes: stand-alone or clustered [203]. The names of these are self descriptive, a single server instance is present in the former and multiple server instances are present in the latter. When JBoss runs in clustered mode a load balancer is placed 'in front of' the application servers to distribute work across the cluster and monitor the cluster for failures (as shown in Figure 5.10. The load balancer is usually part of the cluster itself and is the only part of the cluster the 'client' is aware of. As such, it must be monitored closely as it becomes a single point of failure. Within the cluster (apart from the load balancer) each JBoss node will process some of the requests and sessions will be replicated between the nodes. More specifically, if Stateful Sessions Beans (SFSBs) are being used, the session will be replicated according to some policy defined by the cluster. Often this is after each method invocation. Conversely if Stateless Session Beans (SLSB) are being used then, by definition, the session state does not need to be replicated. This comes at the price of having to re-create the session at each method invocation but with potentially enhanced scalability as the cost of replication grows with the number of nodes in the cluster. We have begun investigating the implications of using JBoss clustering to provide scalability in DECS. In this case, each instance of DECS is actually a cluster of JBoss nodes, each communicating through a HTTP load balancer.

Initial tests show that the scalability is best when the 'Round Robin' policy is used in the load balancer. This policy dispatches requests to each node in turn in a round robin fashion. This makes sense when we consider the fact that when running as a standalone instance, the point of DECS that fails first when heavily loaded is the Thread Pool and maximum number of TCP connections

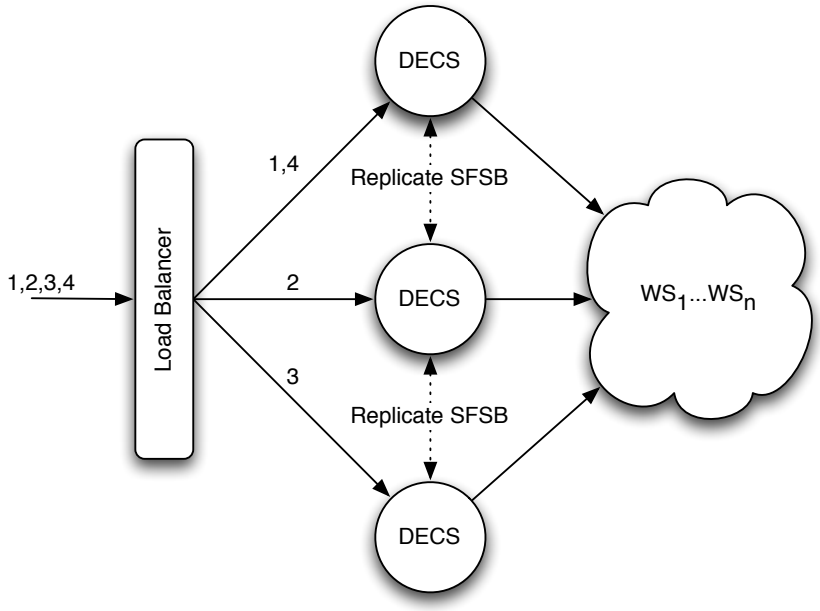


Figure 5.10: JBoss Clustering for DECS

that can be open concurrently. When running many workflows concurrently, or ones that have many parallel tasks, it is possible to hit the limit of the number of Threads available. This happens when each task is a request-response task as the connection must be kept open for the response to received therefore the Thread cannot be put to sleep. When the round robin policy is used each node will share the load approximately equally, allowing the system to scale. It is also possible to increase the number of Threads available to each node. The configuration that comes as standard is fairly conservative and we were able to increase it by 100% before other parts of the system became overloaded.

It should also be noted that the use of clustering with a load balancer also provides a higher availability version of DECS. The load balancer monitors the group for any nodes that have failed and omits them from the next 'round' if it suspects failure. This allows a system administrator to inspect and repair the machine allowing it to re-join the cluster again.

If a replicated database is being used behind DECS, simply replicating the application servers may not provide the scalability required. Maintaining consistency of the database will become more expensive if both the database and application servers are replicated. Although we realise this is a problem, we have not yet investigated a solution.

5.4.9 Workflow Designer

A simple Graphical User Interface has been developed with contribution from Doug Palmer and Ellis Solaiman. The Workflow Designer allows a user to graphically author workflows to be deployed into DECS. It is possible to import template tasks for specific Web Services from the WSDL description of that service. To achieve this the location of the WSDL is entered into the preferences panel and (assuming the WSDL is accessible and well formed) template tasks will appear in the Service Palette on the right hand side. The tasks can then be dragged onto the canvas and further customised. If a process is created in the Workflow Designer it can be expanded by right clicking and selecting "Expand Process" which causes it to open in a new tab. Future work on the Workflow Designer includes runtime monitoring of workflows and "replaying" workflows that have previously executed.

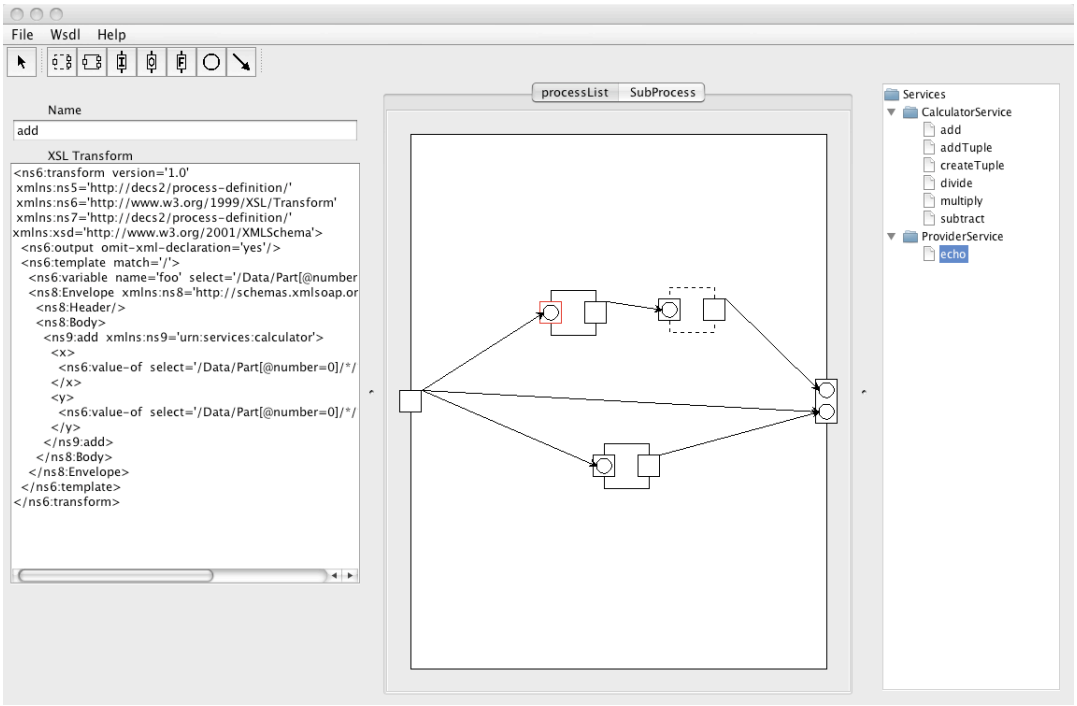


Figure 5.11: The DECS Workflow Designer

5.4.10 Example of Distributed Enactment

Figure 5.12 shows an example of how a very simplistic workflow to generate a business report could be divided to run over two servers, S_a and S_b. Figure 5.12(i) shows the overall workflow with figures 5.12(ii) and 5.12 (iii) showing the two server's views of the workflow.

Server S_a will receive the initial input message for the workflow (gen_report task), instantiate its fragment of the workflow, as shown in 5.12(ii) and inform S_b to do instantiate the part it is responsible for, 5.12(iii). S_a will then perform the get_input task to obtain input from somewhere,

say the user. Following this task, there are no other dependant tasks on this server so $S_a$will communicate with S_b and pass it the data required for S_b to invoke get_data. This particular workflow has been fragmented to minimise data transfer as the get_data task will return a very large data set which must be analysed by the following task. Therefore, the S_b server has been co-located with the database and analysis application. After getting the data and passing it locally to the analyse_data task, there are no more dependant tasks for S_b to perform so it will notify S_a of the results of the analyse_data task. This causes the format_data task to be executed and following this, the report to be sent to the requester. The only data which has been transferred is the initial input data and the raw report data. The large data set produced by the get_data task has only been transferred locally, saving both time and bandwidth. Although this example is simple and can be thought of as hierarchical there is no restriction placed on the structure of the coordination and arbitrary patterns are allowed.

This workflow is also an example of one which is invoked in a one way manner referred to in Section 5.4.3. The client invokes the workflow by sending a message which is received by the ProcessInitiator and the workflow is started. This message will contain WS-A reply to headers which will be used to configure the send_report task to return the results of the workflow to the client.

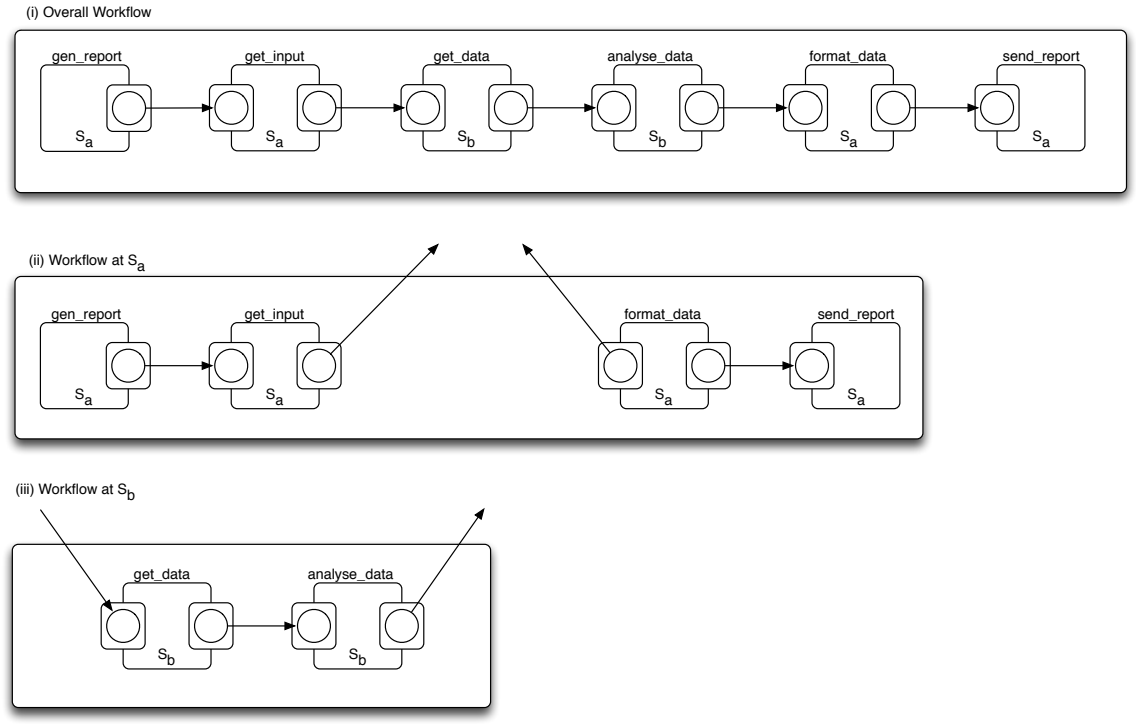


Figure 5.12: Fragmentation of a workflow

5.4.11 Runtime Monitoring of Sequence Constraints

Although this Chapter has focused on the architecture and implementation of a decentralised workflow engine, it is also pertinent to discuss our plans to enforce correspondence to the observable behaviour presented in Chapter 4. We described a method for describing the interaction patterns of services using SSDL and specifically the Sequencing Constraints protocol framework. Sequencing Constraints describe the relative ordering of messages that a service sends and receives. At present it is possible to manually verify that a set of Sequencing Constraints are compatible with each other but there is no runtime support to ensure that a service adheres to them. If we ignore the dynamic binding aspect of Sequencing Constraints we are left with a document that deals purely in terms of messages sent and received by a service. The dynamic binding aspect would be very difficult to check as there are many ways of achieving this even within one programming language let alone the many that are commonly used to implement Web Services. However, it should be possible to verify at runtime that a service does adhere to the Sequencing Constraints specified. One way of achieving this would be to generate a state machine from the Sequencing Constraints and check that an incoming/outgoing message moves the state machine to a valid next state. This could be achieved using interceptors to remove the burden from the service developer with exceptions being raised if illegal messages are discovered. This would also appear to be a promising approach for the monitoring of Service Level Agreements and certainly deserves more investigation.

5.5 Concluding Remarks

We have presented the requirements and abstract architecture for workflow enactment in the context of Virtual Organisations. A key requirement identified was the of flexibility to allow different Virtual Organisations to enact the workflows in different manners according to the business needs. Centralised enactment of business processes does not realise this flexibility and ties the Virtual Organisation to particular enactment styles. Decentralised, peer-to-peer enactment has the potential to allow the required flexibility and we have shown various configurations of workflow engines that might suit different organisational structures while allowing each organisation to maintain their autonomy. With this requirement in mind we have described the implementation details of our prototype decentralised workflow engine, DECS which is based on JavaEE middleware.

Chapter 6

Conclusion

This thesis has presented a programming model for Virtual Organisations using workflow and Web Services technology. We have seen that organisations, both physical and virtual can be defined by the business processes that they implement. Although business processes are valuable intellectual property for an organisation, they are not directly executable. They capture what steps a business takes to achieve a goal in abstract terms that are often technology agnostic. There are many technologies available for representing business processes in an executable format but workflow is one of the most popular. The main advantage of workflow technology for representing business processes is that there is usually a graphical representation of the workflow that has certain semantics and can be related to the business process. The graphical representation means it is easier for people who are not programmers to design, implement and monitor the workflow. It is easier to provide business level abstractions when leveraging workflow technologies than when using other systems such as developing a bespoke application. In addition it is possible for the environment which executes the workflow to provide services and non-functional properties such as fault tolerance, security and provenance.

We have seen that the formation of a Virtual Organisation is one way for physical organisations to interact and collaborate. We defined a Virtual Organisation as "a temporary alliance of organisations that come together to share skills or core competencies and resources in order to better respond to business opportunities". In Chapter 2 we identified a number of possible configurations of Virtual Organisations: Star, market, co, value and parallel alliances, noting that there is not one structure that is a reliable indicator of success. The fact that the structure depends on domain specific requirements is an indicator the flexibility is required in a programming model for Virtual Organisations. Some instances may employ a Trusted Third Party to enact the workflow that defines the VO whereas in some cases decentralised enactment might be more suitable.

Chapter 3 presented the Task Model as a language suitable for the definition of workflows that define the functioning of a Virtual Organisation. We showed the graphical notation, based on the OpenFLOW task model but with the alignment to Web Services rather than Corba and described

the various types of tasks and dependencies. A pattern based evaluation of the Task Model was undertaken and the results show a similar representational capacity to other contemporary workflow languages. In addition to the graphical representation we introduced a π-calculus model that underlies the Task Model notation and showed how it is able to capture the dynamic aspects of workflows. The π-calculus model allows workflows to be analysed for violations of safety and liveness constraints as well as application specific properties. In the future we plan to automate the translation from the XML representation of the Task Model into the π-calculus representation for analysis. In addition we would like to investigate the use of either the Mobility Workbench or ABC model checker for automated analysis of Task Model workflows.

Following on from the Task Model we introduced the SOAP Service Description Language (SSDL) and specifically the Sequencing Constraints protocol framework. SSDL is intended to provide mechanisms for describing the interaction patterns of Web Services and Sequencing Constraints allow the definition of multi-party, multi-message interactions. Interaction patterns have a number of uses within the Virtual Organisation domain. Firstly they can be used to define consistency constraints across multiple services within one organisation. Secondly they can be used to describe the externally visible behaviour of a service provided by an organisation. We have shown how Sequencing Constraints are able to capture dynamic protocols where the participants are not known at design time, a capability not seen in other systems. When the services that are utilised by Task Model workflows define Sequencing Constraints it is possible to ensure that the workflow does not violate those constraints. In addition we showed that it is possible to verify that a set of services exposing Sequencing Constraints are compatible between each other. This behavioural compatibility goes beyond what is currently possible using WSDL and provides an alternative, service centric perspective to that of WS-CDL.

The final Chapter of this thesis has presented the architecture and prototype implementation of a decentralised peer-to-peer workflow engine for Virtual Organisations. The architecture identified the key elements to any such system and the interactions between them. The key element again is flexibility to allow different configurations for different Virtual Organisations. We describe the advantages of the peer-to-peer architecture for domains other than Virtual Organisations. When considering large data sets it might be possible to reduce the network costs associated with transferring large amounts of data by locating a workflow engine close to the data. Another workflow engine may be located elsewhere and used to coordinate other parts of the workflow. Only the data explicitly required by the other workflow engine is transferred rather than transferring the whole data set in a hub and spoke manner. We presented the implementation of DECS, a prototype workflow engine that realises the abstract architecture defined earlier. DECS makes use of the Java 5 Enterprise Edition and is intended to be executed within the JBoss Application Server. We described the features of DECS and gave an example of peer-to-peer execution.

6.1 Future Work

In our previous work we augmented WSDL with Sequencing Constraints Notations [204]. This allowed specification of similar protocols albeit in a slightly different notation. During this work we developed an XSLT stylesheet that converted the WSDL + Sequencing Constraints into Promella that can be analysed using the SPIN model checker[205, 146]. The work with WSDL did not take into account dynamic protocols so the π-calculus mobility was unnecessary. In the future we plan to consider whether an XSLT stylesheet would be a suitable way of transforming the SSDL SC protocols into a format that can be analysed by model checkers such as ABC or MWB.

Current tools for testing the equivalence of services only tend to consider static equivalent rather than dynamic equivalence. By that we mean that they only consider that the services have the same WSDL interface. As we have seen, WSDL does not take into account the relative ordering of messages therefore it is not possible to take this into account when considering equivalence. As the Sequencing Constraints framework (and others) describe the ordering it would be interesting to investigate whether the notion of equivalence becomes more useful. Deciding whether one service 'simulates' the other would appear to be a promising starting point.

An interesting idea is presented in [206] where the authors mine log files to determine the usage patterns for different applications. The use of the derived data is to generate realistic testing scenarios for future versions. However, there is no reason why similar techniques shouldn't be used in combination with Web Services. It would be interesting to look at the log files of existing services and generate proposed SSDL contracts from the derived usage pattens. These could be used by tools to suggest possible contracts or to test contracts against 'live' data.

At present, DECS does not provide as many facilities for the design, monitoring and reconfiguration of workflows as other contemporary workflow engines [121, 207]. In the future we wish to rectify this by providing comprehensive tool support which should include the following: an extension of our existing graphical interface presented in Section 5.4.9. At present the workflow designer is usable but not always intuitive in its working. We wish to provide more 'wizard' like interfaces, more intelligent error reporting and a more comprehensive suite of services that are 'known' to the application. Secondly, we wish to extend the Workflow designer so that it encompasses monitoring facilities too. These should allow the user to start, stop and inspect the state of a workflow on the local or remote machine. Whilst the first two parts will be trivial, the monitoring aspect requires more thought. For the monitoring to work the Workflow Designer requires a method of receiving status updates from DECS. Such a system must be lightweight and able to handle a large number of transactions. If both the Workflow Designer and the DECS instance being monitored are on the same network, it might be possible to use the Java Message Service (JMS). However, when this isn't the case we encounter problems with firewalls at the edges of the network and must look to a more

inter-operable solution. Initially, NaradaBrokering looks promising and appears to offer the features we require.

Any workflow deployed for a significant period of time will need to adapt to take into account services being moved or withdrawn. We hope to offer graphical support in a future version of the Workflow Designer to allow this. It should only be possible to redesign a workflow which is in a consistent state, i.e. no tasks are active [208], and only the parts which are yet to execute should be refactored. However, there is no reason why the state of the workflow should not be persisted, reconfigured and started again. Indeed the current DECS architecture makes this a relatively straightforward addition. Instead of putting messages onto the notification queue, these can be saved into a database. This ensures that no new tasks will begin execution and when all executing tasks have reached a completed state the workflow can be adapted. Once the user has completed the redesign, the messages that were saved to the database are placed onto the notification queue and the workflow will start again.

In addition to extending the feature set of DECS and the Workflow Designer, we wish to run more comprehensive performance analysis of the implementation. It would be interesting to compare the performance with other distributed workflow engines such as jOpera. However, as we discussed earlier, jOpera is optimised for cluster based computing whereas DECS is targeted at larger scale networks with higher latency and message loss. We expect jOpera to perform better on networks with high bandwidth but anticipate it being problematic to install over a wider area network.

Bibliography

[1] Luis M. Camarinha-Matos and Hamideh Afsarmanesh. The Emerging Discipline of Collaborative Networks. In *Enterprises and Collaborative Networks*. Kluwer Academic Publisher, 2004.

[2] J. Halliday, S. K. Shrivastava, and S. M. Wheater. Flexible Workflow Management in the OPENflow system. In *Proc of the 5th Int. Enterprise Distributed Object Computing Conference (EDOC 2001), Seattle*, 2001.

[3] S Parastatidis and J Webber. The SOAP Service Description Language. http://www.ssdl.org/docs/v1.3/html/SSDL%20v1.3.html, April 2005.

[4] Lucas D. Introna, Hope Moore, and Mike Cushman. The Virtual Organisation - Technical or Social Innovation? Lessons from the Film Industry. Technical Report 72, London School of Economics, 1999.

[5] S. Goldman, R. Nagel, and K. Preiss. *Agile competitors and virtual organisations*. Van Nostrand Reinhold, 1995.

[6] J Byrne. The Virtual Corporation. *Business Week*, pages 98–102, February 1993.

[7] Philip Kraft and Duane P. Truex. Postmodern Management and Information Technology in the Modern Industrial Corporation. In *Proceedings of the IFIP WG8.2 Working Conference on Information Technology and New Emergent Forms of Organizations*, pages 113–127, Amsterdam, The Netherlands, The Netherlands, 1994. North-Holland Publishing Co.

[8] Shawn Tully and Tricia Welsh. The Modular Corporation. *Fortune*, 127(3):p106 – 115, 02 1993.

[9] R. Johnston and P.R. Lawrence. Beyond vertical integration – The rise of the valueadding partnership. *Harvard Business Review*, 66(4):94–101, 1988.

[10] G. Morgan. *Creative organization theory. A resourcebook*. London: SAGE Publications, 1989.

[11] Timothy J. Norman, Alun Preece, Stuart Chalmers, Nicholas R. Jennings, Michael Luck, Viet D. Dang, Thuc D. Nguyen, Vikas Deora, Jianhua Shao, W. Alex Gray, and Nick J. Fiddian. Agent-based formation of virtual organisations. *Knowledge-Based Systems*, 17(2-4):103–111, 2004.

[12] Nick Lethbridge. An I-Based Taxonomy of Virtual Organisations and the Implications for Effective Management. *Informing Science The International Journal of an Emerging Transdiscipline*, 4(1):17–24, 2001.

[13] Daniel E. O'Leary, Daniel Kuokka, and Robert Plant. Artificial intelligence and virtual organizations. *Commun. ACM*, 40(1):52–59, 1997.

[14] J. W. Bryans, J.S. Fitzgerald, C.B. Jones, and I Mozolevsky. Formal Modelling of Dynamic Coalitions, with an Application in Chemical Engineering. In *IEEE-ISoLA 2006. Second International Symposium on Leveraging Applications of Formal Methods , Verification and Validation*, pages 90–97. IEEE, 2006.

[15] N. Galeano, D. Guerra, J. Irigoyen, and A. Molina. Virtual Breeding Environment: A First Approach to Understand Working and Sharing Principles. In *Proceedings of First International Conference in Interoperability of Enterprise Software and Applications*, Geneva, Switzerland, 21 - 25 February, 2005.

[16] Ovidiu Sever Noran. Towards a Meta-methodology for Collaborative Networked Organisations. In Luis M.Camarinha-Matos, editor, *IFIP 18th World Computer Congress / 5th Working Conference on VIRTUAL ENTERPRISES (PRO-VE05)*. Kluwer Academic Publishers, 2004.

[17] M. Gruber and M. Noester. Investigating Structural Settings of Virtual Organisations. In *Proceedings of the 11th International Conference on Concurrent Enterprising: Integrated Engineering of Products, Services and Organisations*, pages 245 – 251, June 2005.

[18] Daniel E. O'Leary. Virtual organizations: two choice problems. In *ICIS '98: Proceedings of the international conference on Information systems*, Atlanta, GA, USA, 1998.

[19] Hisup Park, Jay M. Tenenbaum, and Rick Dove. Agile Infrastructure for Manufacturing Systems (AIMS) - A vision for Transforming the US Manufacturing Base. In *Proceedings Defense Manufacturing Conference*, 1993.

[20] W.M.P. van der Aalst and M. Weske. The P2P approach to Interorganizational Workflows. In *Proceedings of the 13th International Conference on Advanced Information Systems Engineering (CAiSE'01)*, volume 2068 of *Lecture Notes in Computer Science*, pages 140–156. Springer-Verlag, 2001.

[21] W3C. Web Services Choreography Description Language (WS-CDL) Version 1.0. http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/, 2004.

[22] C. Molina-Jimenez, S. K. Shrivastava, and J. Warne. A Method for Specifying Contract Mediated Interactions. In *EDOC '05: Proceedings of the Ninth IEEE International EDOC Enterprise Computing Conference*, pages 106–118, Washington, DC, USA, 2005. IEEE Computer Society.

[23] M.L. Emiliani and D.J. Stec. Online reverse auction purchasing contracts. *Supply Chain Management: An International Journal*, 6(3):101 – 105, 2001.

[24] C Zirpins and W Emmerich. A reference model of virtual service production networks. *Service Oriented Computing and Applications*, 2(2):145–166, 2008.

[25] C. Zirpins and W. Emmerich. Service-oriented modelling and design of virtual business services. research notes, Dept. of Computer Science, University College London, 2008.

[26] P Periorellis, N Cook, H Hiden, A Conlin, M.D. Hamilton, J Wu, J Bryans, X Gong, P Watson, R Smith, and A Wright. GOLD Infrastructure for Virtual Organizations. In *5th All Hands Meeting, AHM2006*, 2006.

[27] Microsoft Corporation. Microsoft Passport. http://www.passport.net.

[28] Liberty Alliance. http://www.projectliberty.org/.

[29] C. Molina-Jimenez, S. K. Shrivastava, E. Solaiman, and J. Warne. Contract representation for run-time monitoring and enforcement. *E-Commerce, 2003. CEC 2003. IEEE International Conference on*, pages 103–110, 24-27 June 2003.

[30] P. F. Robinson, N. Cook, and S. K. Shrivastava. Implementing fair non-repudiable interactions with Web services. In *EDOC Enterprise Computing Conference, 2005 Ninth IEEE International*, pages 195–206, 2005.

[31] eBay Inc. eBay. www.ebay.co.uk.

[32] N. Nayak, T. Chao, J. Li, J. Mihaeli, R. Das, A. Derebail, and J Soo Hoo. Role of Technology in Enabling Dynamic Virtual Enterprises. In *Proceedings of International Workshop on Open Enterprise Solutions: Systems, Experiences, and Organizations*, 2001.

[33] OASIS. ebxml. http://www.ebxml.org/.

[34] RosettaNet Corporation. RosettaNet. http://www.rosettanet.org.

[35] Suresh Damodaran. B2B integration over the Internet with XML: RosettaNet successes and challenges. In *WWW Alt. '04: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 188–195, New York, NY, USA, 2004. ACM.

[36] RosettaNet Corporation. RosettaNet Implementation Framework. http://www.rosettanet.org/cms/sites/RosettaNet/Standards/RStandards/rnif/.

[37] Rania Khalaf. From RosettaNet PIPs to BPEL processes: A three level approach for business protocols. *Data Knowl. Eng.*, 61(1):23–38, 2007.

[38] G. Piccinelli, A. Finkelstein, and E. Stammers. Automated Engineering of e-business Processes: the RosettaNet case study. In *6th International Conference on Systems, Cybernetics and Informatics*, Orlando, Florida, 2002. ICSCI.

[39] C. Molina-Jimenez, S. K. Shrivastava, and S. J. Woodman. On State Synchronization of Business Conversations. In *Proceedings of the 8th IEEE International Conference on E-Commerce and Technology (CEC 2006)*, pages 324–327. IEEE Computer Society, 2006.

[40] OASIS. Reference model for service oriented architecture v 1.0. http: //www.oasis-open.org/committees/soa- rm/, 2006.

[41] D. Sprott and L. Wilkes. Understanding service-oriented architecture. *Microsoft Architects Journal*, 1(1):10–17, 2004.

[42] M. Endrei, J. Ang, A. Arsanjani, S. Chua, P. Comte, P. Krogdahl, M. Luo, and T. Newling. *Patterns: Service Oriented Architecture And Web Services.* IBM Redbooks, 2004.

[43] T. Erl. *Service-Oriented Architecture : A Field Guide to Integrating XML and Web Services.* Prentice-Hall, 2004.

[44] T. Erl. *Service-Oriented Architecture (SOA): Concepts, Technology, and Design.* Prentice-Hall, 2005.

[45] M. P. Singh and M. N. Huhns. *Service-Oriented Computing: Semantics, Processes, Agents.* Wiley, 2005.

[46] Deborah J. Armstrong. The quarks of object-oriented development. *Commun. ACM*, 49(2):123–128, 2006.

[47] D. Pallmann. *Programming INDIGO.* Microsoft Press, 2005.

[48] P. Greenfield. Service-oriented architectures and technologies. In *Essential Software Architecture*, pages 217–237. Springer, 2006.

[49] J. Webber and S. Parastatidis. Realising service oriented architectures using web services. In *Service Oriented Computing*. MIT Press, (In press).

[50] D. Box. A guide to developing and running connected systems with indigo. *MSDN Magazine*, 19(1), 2004.

[51] D. Kaye. *Loosely Coupled: The Missing Pieces of Web Services*. RDS Press, 2003.

[52] Momentum SI. Loose Coupling. http://www.serviceoriented.org/coupling.html.

[53] J Webber and S Chatterjee. *Developing Enterprise Web Services: An Architects Guide*. Prentice-Hall, 2003. ISBN: 0131401602.

[54] Carlos E. Perez. SOA Principles and Modularity. http://www.manageability.org/blog/stuff/soa-rediscovering-modularity.

[55] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, March 2000.

[56] Object Management Group. CORBA/IIOP specification. http://www.omg.org/technology/documents/formal/corba_iiop.htm.

[57] Microsoft Corporation. Distributed Component Object Model (DCOM). http://www.microsoft.com/com/.

[58] Sun Microsystems. Java remote method invocation (Java RMI). http://java.sun.com/products/jdk/rmi/.

[59] V. Matena, S. Krishnan, L. DeMichiel, and B. Stearns. *Applying Enterprise JavaBeans 2.1: Component-Based Development for the J2EE Platform (2nd Edition)*. Addison-Wesley Professional, 2003.

[60] G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Profes- sional, 2003.

[61] G. Banavar, T. Chandra, R. Strom, and D. Sturman. A case for message oriented middleware. In *13th International Symposium on Distributed Computing (DISC)*, pages 846–863, 1999.

[62] S. Vinoski. RPC under fire. *IEEE Internet Computing*, 9(5):93–95, 2005.

[63] W. Vogels. Web services are not distributed objects. *IEEE Internet Computing*, 7(6):59–66, 2003.

[64] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A note on distributed computing. Technical report, Sun Microsystems Laboratories, 1994.

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

[66] Francisco Curbera, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi, and Sanjiva Weerawarana. Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, 6(2):86–93, 2002.

[67] Paul Fremantle, Sanjiva Weerawarana, and Rania Khalaf. Enterprise services. *Commun. ACM*, 45(10):77–82, 2002.

[68] L. F. Cabrera, C. Kurt, and D. Box. An introduction to the web services architecture and its specifications. *Microsoft Web Services Technical Articles*, 2004.

[69] W3C. Web services architecture. http://www.w3.org/TR/ws- arch/, 2004.

[70] S. Vinoski. Web services interaction models. current practice. *IEEE Internet Computing*, 6(3):89–91, 2002.

[71] Microsoft Corporation. Web services and the microsoft platform. http://msdn.microsoft.com/webservices/default.aspx?pull=/library/en-us/dnwebsrv/html/wsmsplatform.asp.

[72] InnoQ. Web Services Standards Poster. http://www.innoq.com/resources/ws-standards-poster/.

[73] Steve Vinoski. WS-Nonexistent Standards. *IEEE Internet Computing*, 8(6):94–96, 2004.

[74] Sun Microsystems. Web services essentials. http://java.sun. com/webservices/, 2006.

[75] Microsoft Corporation. Web services specifications. http://msdn. microsoft.com/webservices/webservices/understanding/specs/, 2006.

[76] IBM developerWorks. Standards and web services. http://www- 128. ibm.com/developerworks/webservices/standards/, 2006.

[77] W3C. SOAP version 1.2 part 1: Messaging framework. http://www. w3.org/TR/soap12-part1/, 2003.

[78] S. Loughran and E. Smith. Rethinking the Java SOAP stack. Technical Report HPL-2005-83, Hewlett-Packard Bristol Laboratories, 2005.

[79] W3C. Web services addressing (WS-Addressing). http://www.w3.org/ Submission/ws- addressing, 2004.

[80] K Ballinger, D Ehnebuske, M Gudgin, M Nottingham, and P Yendluri. WS-I Basic Profile Version 1.0, April 2004.

[81] James Pasley. How BPEL and SOA Are Changing Web Services Development. *IEEE Internet Computing*, 9(3):60–67, 2005.

[82] W3C. XML schema. http://www.w3.org/XML/Schema, 2004.

[83] S. Vinoski. Putting the "web" into web services. web services interaction models, part 2. *IEEE Internet Computing*, 6(4):90–92, 2002.

[84] Uwe Zdun, Markus Völter, and Michael Kircher. Pattern-Based Design of an Asynchronous Invocation Framework for Web Services. *Int. J. Web Service Res.*, 1(3):42–62, 2004.

[85] W3C. Web services description language (WSDL) version 2.0 part 1: Core language. http://www.w3.org/TR/wsdl20/, 2007.

[86] R. Salz. WSDL 2: Just say no. http://webservices.xml.com/pub/a/ ws/2004/11/17/salz.html, 2004.

[87] D. Hinchcliffe. Web service description languages: When there is nothing left to take away. http://hinchcliffe.org/archive/2005/05/10/ 215.aspx, 2005.

[88] Jboss Corporation. Jboss Web Services. http://labs.jboss.com/jbossws/.

[89] L. G. Meredith and Steve Bjorg. Contracts and types. *Communinications of the ACM*, 46(10):41–47, 2003.

[90] Object Management Group. IDL syntax and semantics chapter. http: //www.omg.org/cgi-bin/doc?formal/02- 06- 39.

[91] S. Parastatidis, S. J. Woodman, J. Webber, D. Kuo, and P. Greenfield. Asynchronous messaging between web services using SSDL. *IEEE Internet Computing*, 10(1):26–39, 2006.

[92] Nirmit Desai, Ashok U. Mallya, Amit K. Chopra, and Munindar P. Singh. Interaction Protocols as Design Abstractions for Business Processes. *IEEE Trans. Softw. Eng.*, 31(12):1015–1027, 2005.

[93] Johannes Maria Zaha, Marlon Dumas, Arthur H. ter Hofstede, Alistair P. Barros, and Gero Decker. Service Interaction Modeling: Bridging Global and Local Views. In *10th IEEE International Enterprise Distributed Object Computing Conference (EDOC'06)*, 2006.

[94] Business Process Execution Language for Web Services (BPEL4WS) version 1.1. http://www.ibm.com/developerworks/library/specification/ws-bpel/.

[95] S. Parastatidis, J. Webber, S. J. Woodman, D. Kuo, and P. Greenfield. An Introduction to the SOAP Services Decsription Language. Technical Report CS-TR-898, University of Newcastle, 2005.

[96] Boualem Benatallah, Fabio Casati, and Farouk Toumani. Representing, analysing and managing web service protocols. *Data Knowl. Eng.*, 58(3):327–357, 2006.

[97] A. Brogi, C. Canal, E. Pimentel, and A. Vallecillo. Formalizing Web Service Choreographies. In *Proc. of First International Workshop on Web Services and Formal Methods*, 2004.

[98] W3C. OWL-S: Semantic Markup for Web Services. http://www.w3.org/Submission/OWL-S, 2004.

[99] Alistair P. Barros, Marlon Dumas, and Arthur H.M. ter Hofstede. Service Interaction Patterns. In *3rd International Conference on Business Process Management*, 2005.

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

[101] M. Carbone, K. Honda, N. Yoshida, R. Milner, G. Brown, and S. Ross-Talbot. A theoretical basis of communication-centred concurrent programming. http://www.w3.org/2002/ws/chor/edcopies/theory/note.pdf, 2006.

[102] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured Communication-Centred Programming for Web Services. *Programming Languages and Systems*, pages 2–17, 2007.

[103] Geguang Pu, Jianqi Shi, Zheng Wang, Lu Jin, Jing Liu, and Jifeng He. The Validation and Verification of WS-CDL. In *APSEC '07: Proceedings of the 14th Asia-Pacific Software Engineering Conference*, pages 81–88, Washington, DC, USA, 2007. IEEE Computer Society.

[104] Hongli Yang, Xiangpeng Zhao, Zongyan Qiu, Geguang Pu, and Shuling Wang. A Formal Model for Web Service Choreography Description Language (WS-CDL). In *ICWS '06: Proceedings of the IEEE International Conference on Web Services*, pages 893–894, Washington, DC, USA, 2006. IEEE Computer Society.

[105] Jan Mendling and Michael Hafner. From Inter-organizational Workflows to Process Execution: Generating BPEL from WS-CDL. *On the Move to Meaningful Internet Systems 2005: OTM Workshops*, pages 506–515, 2005.

[106] Gero Decker, Johannes Zaha, and Marlon Dumas. Execution Semantics for Service Choreographies. In *Web Services and Formal Methods*, pages 163–177, 2006.

[107] Johannes Maria Zaha, Alistair P. Barros, Marlon Dumas, and Arthur H ter Hofstede. Let's Dance: A Language for Service Behavior Modeling. Technical report, Queensland University of Technology, 2006.

[108] Eleanna Kafeza, Dickson K. W. Chiu, and Irene Kafeza. View-Based Contracts in an E-Service Cross-Organizational Workflow Environment. In *TES '01: Proceedings of the Second International Workshop on Technologies for E-Services*, pages 74–88, London, UK, 2001. Springer-Verlag.

[109] Jie Meng, Stanley Y.W. Su, Herman Lam, Abdelsalam Helal, Jingqi Xian, Xiaoli Liu, and Seokwon Yang. DynaFlow: a dynamic inter-organisational workflow management system. *International Journal of Business Process Integration and Management*, 1:101–115(15), 8 June 2006.

[110] Workflow Management Coalition. The Workflow Reference Model Version 1.1, January 19th 1995.

[111] Fabio Casati and Angela Discenza. Supporting workflow cooperation within and across organizations. In *SAC 2000: Proceedings of the 2000 ACM symposium on Applied computing*, pages 196–202, New York, NY, USA, 2000. ACM.

[112] Workflow Management Coalition. Conformance to WfMC Reference Model. http://www.wfmc.org/standards/conformance_chart.htm.

[113] Workflow Management Coalition. XPDL. http://www.wfmc.org/standards/xpdl.htm.

[114] BPMI. Business Process Modelling Notation. http://www.bpmn.org/, 2004.

[115] Chun Ouyang, M. Dumas, Ter, and W. M. P. van der Aalst. From BPMN Process Models to BPEL Web Services. In *International Conference on Web Services, ICWS '06.*, pages 285–292, 2006.

[116] SOA Tools BPMN Modeller. http://www.eclipse.org/stp/bpmn/.

[117] Orbus Software. BPMN Tools for Visio. http://www.bpmn.co.uk/.

[118] W3C. XML Path Language (XPATH). http://www.w3.org/TR/xpath.

[119] L. Baresi, A. Maurino, and S. Modafferi. Towards distributed bpel orchestrations. In *Electronic Communications of the EASST*, 2006.

[120] G Chafle, S Chandra, V Mann, and M Gowri Nanda. Decentralized orchestration of composite web services. In *Alternate track papers & posters of the 13th international conference on World Wide Web*, pages 134–143. ACM Press, 2004.

[121] Cesare Pautasso and Gustavo Alonso. JOpera: A Toolkit for Efficient Visual Composition of Web Services. *Int. J. Electron. Commerce*, 9(2):107–141, 04-5.

[122] C. Pautasso, T. Heinis, and G. Alonso. Autonomic execution of Web service compositions. *Web Services, 2005. ICWS 2005. Proceedings. 2005 IEEE International Conference on*, pages –442, 11-15 July 2005.

[123] B Benatallah, Q Z. Sheng, and M Dumas. The Self-Serv Environment for Web Services Composition. *IEEE Internet Computing*, pages 40–48, July 2003.

[124] Boualem Benatallah, Marlon Dumas, Quan Z. Sheng, and Anne H.H. Ngu. Declarative Composition and Peer-to-Peer Provisioning of Dynamic Web Services. In *ICDE '02: Proceedings of the 18th International Conference on Data Engineering*, page 297, Washington, DC, USA, 2002. IEEE Computer Society.

[125] Quan Z. Sheng, Boualem Benatallah, Marlon Dumas, and Eileen Oi-Yan Mak. SELF-SERV: a platform for rapid composition of web services in a peer-to-peer environment. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 1051–1054. VLDB Endowment, 2002.

[126] Wil van der Aalst and Maja Pesic. DecSerFlow: Towards a Truly Declarative Service Flow Language. In Frank Leymann, Wolfgang Reisig, Satish R. Thatte, and Wil van der Aalst, editors, *The Role of Business Processes in Service Oriented Architectures*, number 06291 in Dagstuhl Seminar Proceedings, 2006.

[127] D. Giannakopoulou and K. Havelund. Automata-Based Verification of Temporal Properties on Running Programs. In *16th IEEE International Conference on Automated Software Engineering*, pages 412–416. IEEE Computer Society Press, 2001.

[128] R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. Simple On-The-Fly Automatic Verification of Linear Temporal Logic. In *Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification*, 1996.

[129] P. Grefen, K. Aberer, Y. Hoffner, and H. Ludwig. CrossFlow: Cross-Organizational Workflow Management in Dynamic Virtual Enterprises. *International Journal of Computer Systems Science and Engineering*, 15(5):277–290, 2000.

[130] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R. Pocock, Anil Wipat, and Peter Li. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics Journal*, 20(17):3045–3054, 2004.

[131] Janaka Balasooriya, Mohini Padhye, Sushil K. Prasad, and Shamkant B. Navathe. BondFlow: A System for Distributed Coordination of Workflows over Web Services. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 1*, page 121.1, Washington, DC, USA, 2005. IEEE Computer Society.

[132] L. Lamport. Proving the Correctness of Multiprocess Programs. *Software Engineering, IEEE Transactions on*, SE-3(2):125–143, March 1977.

[133] Gero Decker and Mathias Weske. Behavioral Consistency for B2B Process Integration. *Advanced Information Systems Engineering*, pages 81–95, 2007.

[134] Axel Martens. Analyzing Web Service Based Business Processes. *Fundamental Approaches to Software Engineering*, pages 19–33, 2005.

[135] Carlos Canal, Ernesto Pimentel, and José M. Troya. Compatibility and inheritance in software architectures. *Sci. Comput. Program.*, 41(2):105–138, 2001.

[136] Soffer P. and Ghattas J. Business Process Flexibility in Virtual Organizations. In *Workshop on Business Process Modeling, Design and Support (BPMDS'06)*, pages 188–197, 2006.

[137] Boualem Benatallah, Fabio Casati, and Farouk Toumani. Web service conversation modeling: A cornerstone for e-business automation. *IEEE Internet Computing*, 8(1):46–54, 2004.

[138] F. van Breugel and M. Koshkina. Models and Verification of BPEL. Technical report, York University, http://www.cse.yorku.ca/ franck/research/drafts/tutorial.pdf, 2006.

[139] Karsten Schmidt and Christian Stahl. A Petri net semantic for BPEL4WS - validation and application. In Ekkart Kindler, editor, *Proceedings of the 11th Workshop on Algorithms and Tools for Petri Nets (AWPN'04)*, pages 1–6, oct 2004.

[140] A. Martens. On compatibility of web services. *Petri Net Newsletter*, 65:12–20, 2003.

[141] A. Martens. Consistency between executable and abstract processes. In *Proceedings of the 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service*, pages 60–67. IEEE, 2005.

[142] S. Hinz, K. Schmidt, and C. Stahl. Transforming BPEL to Petri nets. In W.M.P. van der Aalst, B. Benatallah, F. Casati, and F. Curbera, editors, *Proceedings of the 3rd International Conference on Business Process Management*. Springer-Verlag, 2005.

[143] K. Schmidt. LoLA – a low level analyser. In M. Nielsen and D. Simpson, editors, *Proceedings of the 21st International Conference on Application and Theory of Petri Nets*, volume 1825, pages 465–474, 2000.

[144] H.M.W. Verbeek and W.M.P. van der Aalst. Analyzing BPEL processes us- ing Petri nets. In D. Marinescu, editor, *Proceedings of the Second International Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management*, pages 59–78, 2005.

[145] C. Ouyang, W.M.P. van der Aalst, S. Breutel, M. Dumas, A.H.M. ter Hofstede, and H.M.W. Verbeek. Formal semantics and analysis of control flow in WS-BPEL. Technical report, BPM Center, 2005.

[146] G Holzmann. *The SPIN model checker: Primer and reference manual*. Addison Wesley, 2003.

[147] X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL web services. In *Proceedings of the 13th International World Wide Web Conference*, pages 621–630, 2004.

[148] S. Nakajima. Model-checking behavioral specification of BPEL applications. In *Proceedings of the International Workshop on Web Languages and Formal Meth- ods*, volume 1512), pages 89–105, 2005.

[149] J. Magee and J. Kramer. *Concurrency: State Models and Java Programs*. Wiley, 2 edition, 2006.

[150] H. Foster. A Rigorous Approach To Engineering Web Service Compositions. Phd thesis, Imperial College, London, UK, January 2006.

[151] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service compositions. In *Proceedings of 18th IEEE International Conference on Automated Software Engineering*, pages 152–163. IEEE, 2003.

[152] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based analysis of obligations in web service choreography. In *Proceedings of the International Conference on Internet and Web Applications and Services*. IEEE, 2006.

[153] Howard Foster, Wolfgang Emmerich, Jeff Kramer, Jeff Magee, David Rosenblum, and Sebastian Uchitel. Model checking service compositions under resource constraints. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 225–234, New York, NY, USA, 2007. ACM.

[154] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.

[155] R. Milner. The Polyadic pi-Calculus: A Tutorial. *Logic and Algebra of Specification*, 1993.

[156] Riccardo Pucella. Review of The Pi-Calculus: A Theory of Mobile Processes. *SIGACT News*, 34(1), 2003.

[157] R Milner, J Parrow, and D Walker. A Calculus of Mobile Processes. *Information and Computation*, 100:1–40, 1992.

[158] D. Sangiorgi. A Theory of Bisimulation for the π-calculus. *Acta Informatica*, 33:69–97, 1996. An extract appeared in *Proc. CONCUR '93*, Lecture Notes in Computer Science 715, Springer Verlag.

[159] Nadia Busi, Roberto Gorrieri, Claudio Guidi, Roberto Lucchi, and Gianluigi Zavattaro. Choreography and Orchestration: A Synergic Approach for System Design. In *International Conference on Service Oriented Computing*, pages 228–240, 2005.

[160] Sun Microsystems. Java Enterprise Edition. http://java.sun.com/javaee/.

[161] Jboss Corporation. JBoss Application Server. http://www.jboss.org.

[162] Sun Microsystems. Glassfish - Open Source Application Server. https://glassfish.dev.java.net/.

[163] IBM. Websphere Application Server. http://www-306.ibm.com/software/webservers/appserv/was/.

[164] Oracle Corporation. http://www.oracle.com/appserver/index.html.

[165] E Christensen, F Curbera, G Meredith, and S Weerawarana. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl.

[166] Knowledge Based Systems, Inc. IDEF0. http://www.idef.com/IDEF0.html.

[167] Object Management Group. Unified Modelling Language. http://www.uml.org.

[168] ObjectWeb. What is Middleware? http://middleware.objectweb.org/.

[169] Sun Microsystems. The Java VM Specification. http://tinyurl.com/2kyta4.

[170] S. M. Wheater, S. K. Shrivastava, and F. Ranno. A CORBA Compliant Transactional Workflow System for Internet Applications. In *Proc. Int'l Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, September 1998.

[171] Duncan Hull, Robert Stevens, Phillip Lord, Chris Wroe, and Carole Goble. Treating shimantic web syndrome with ontologies. In *First AKT workshop on Semantic Web Services (AKT-SWS04) KMi, The Open University, Milton Keynes, UK. December 8, 2004*, 2004.

[172] Sylvia C. Wong, Simon Miles, Weijian Fang, Paul Groth, and Luc Moreau. Provenance-based validation of e-science experiments. In *4th Internation Semantic Web Conference (ISWC'05)*, 2005.

[173] W. van der Aalst and A. Hofstede. YAWL: Yet Another Workflow Language. Technical report, Queensland University of Technology, 2002.

[174] Petia Wohed, Wil M. P Aalst, Marlon Dumas, and Arthur H. M. Ter Hofstede. Pattern Based Analysis of BPEL4WS. Technical report, Queensland University of Technology, 2002.

[175] Leslie Lamport. A New Approach to Proving the Correctness of Multiprocess Programs. *ACM Trans. Program. Lang. Syst.*, 1(1):84–97, 1979.

[176] Robin Milner. *A Calculus of Communicating Systems.* Springer Verlag, ISBN 0-387-10235-3, 1980.

[177] CAR Hoare. *Communicating Sequential Processes.* Prentice Hall International, 1985.

[178] T. Murata. Petri nets: Properties, analysis and applications. In *IEEE*, volume 77, pages 541–580, 1989.

[179] RosettaNet Corporation. RosettaNet Partner Interface Protocol Specifications. http://www.rosettanet.org.

[180] Ernst W. Mayr. An algorithm for the general Petri net reachability problem. In *STOC '81: Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 238–246, New York, NY, USA, 1981. ACM.

[181] The Mobility workbench. http://www.it.uu.se/research/group/mobility/mwb.

[182] Sébastien Briais. ABC - Another Bi-Simulation Checker. http://lampwww.epfl.ch/ sbriais/abc/abc.html.

[183] W. M. P. Van Der Aalst. Inheritance of Interorganizational Workflows: How to Agree to Disagree Without Loosing Control? *Inf. Technol. and Management*, 4(4):345–389, 2003.

[184] Formal Systems (Europe) Ltd. Failure Divergence Refinement: FDR2 User Manual. http://www.fsel.com/documentation/fdr2/.

[185] S Parastatidis and J Webber. CSP SSDL Protocol Framework. http://www.ssdl.org/docs/v1.3/html/CSPhtml, April 2005.

[186] OASIS. Web services Context (WS-CTX). www.iona.com/devcenter/standards/WS-CAF/WSCTX.pdf.

[187] OASIS. Web Services Security (WS-Security). http://www.oasis-open.org/committees/wss.

[188] S Parastatidis and J Webber. MEP SSDL Protocol Framework. http://www.ssdl.org/docs/v1.3/html/MEPhtml, April 2005.

[189] D Kuo, P Greenfield, S Parastatidis, and J Webber. Rules-based SSDL Protocol Framework. http://www.ssdl.org/docs/v1.3/html/Rules3.html, April 2005.

[190] S. J. Woodman, S. Parastatidis, and J. Webber. Sequencing Constraints SSDL Protocol Framework. Technical Report CS-TR-903, University of Newcastle, 2005.

[191] Aaron Skonnard. Contract-First Service Development. *MSDN Magazine*, May, 2005.

[192] Savas Parastatidis. Explaining MEST. http://savas.parastatidis.name/2005/01/29/544a6902-40e1-47e8-a51c-18776f3dd036.aspx.

[193] S. J. Woodman, D. J. Palmer, S. K. Shrivastava, and S. M. Wheater. Notations for the Specification and Verification of Composite Web Services. In *Proc of the 8th IEEE International Enterprise Distributed Object Computing Conference*, September 2004.

[194] Sun Microsystems. Enterprise Java Beans. http://java.sun.com/products/ejb/.

[195] V Sarkar. Automatic partitioning of a program dependence graph into parallel tasks. *IBM Journal of Research and Development*, 35:779, 1991.

[196] S. J. Woodman, D. J. Palmer, S. K. Shrivastava, and S. M. Wheater. DECS: A System for Decentralised Coordination of Web Services. In *Proceedings of the first Middleware for Web Services Workshop in association with the 9th IEEE/OMG EDOC Conference*, September 2005.

[197] V. Issarny, F. Tartanoglu, Romanovsky A., and N Levy. Coordinated forward error recovery for composite Web services. In *22nd International Symposium on Reliable Distributed Systems*, pages 167–176, 2003.

[198] Jboss Corporation. JBoss Deployment Architecture. http://tinyurl.com/5zcz2n.

[199] Wikipedia.org. WAR File Format. http://en.wikipedia.org/wiki/WAR_(file_format).

[200] Sun Microsystems. What is a Message Driven Bean (J2EE Tutorial). http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/EJBConcepts5.html.

[201] The Innodb Storage Engine. http://dev.mysql.com/doc/mysql/en/innodb.html.

[202] W3C. Introduction to HTTP Status Codes. http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html.

[203] Jboss Corporation. JBoss Clustering Guide. http://tinyurl.com/6rkl8s.

[204] R. Jimenez-Peris, M. Patino-Martinez, S. J. Woodman, S. K. Shrivastava, D. J. Palmer, S. M. Wheater, B. Kemme, and G. Alonso. D6: Service Specification Language. Technical report, ADAPT Project, 2003.

[205] D. J. Palmer and S. J. Woodman. D8: Analysis Tool. Technical report, ADAPT Project, 2004.

[206] M. De Barros, J. Shiau, Chen Shang, K. Gidewall, Hui Shi, and J. Forsmann. Web Services Wind Tunnel: On Performance Testing Large-Scale Stateful Web Services. *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on*, pages 612–617, June 2007.

[207] Active Endpoints. ActiveBPEL. http://www.activevos.com/community-open-source.php.

[208] S. K. Shrivastava and S. M. Wheater. Architectural Support for Dynamic Reconfiguration of Large Scale Distributed Applications. In *Proceedings of The 4th International Conference on Configurable Distributed Systems (CDS'98)*, Annapolis, Maryland, USA, 1998.

Appendix A

Complete Formalisation of the Seller's Workflow

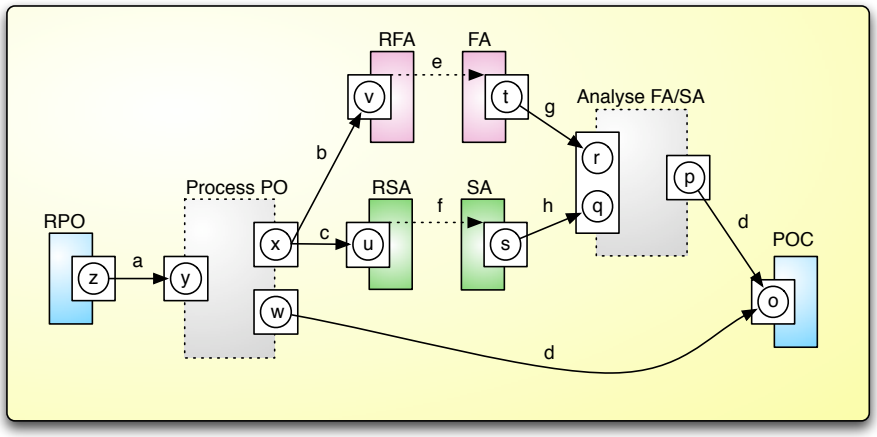


Figure A.1: Seller's Workflow Showing Labels for parts and dependency groups

A.1 Request Purchase Order (RPO)

$$RPO = (\nu\ iwt,\ iwd,\ nit,\ nid,\ owt,\ owd,\ not,\ nod) \tag{A.1}$$

$$(iwt \mid iwd).(nit \mid nid).\tau_{interact}.(owt \mid owd).(not \mid nod) \tag{A.2}$$

$$\mid InputWaitTemp(iwt) \mid InputWaitData(iwd) \tag{A.3}$$

$$\mid NotifyInputTemporal(nit) \mid NotifyInputData(nid) \tag{A.4}$$

$$\mid OutputWaitTemporal(owt) \mid OutputWaitData(owd) \tag{A.5}$$

$$\mid NotifyOutputTemporal(nod) \mid NotifyOutputData(nod) \tag{A.6}$$

$$InputWaitTemporal(r) = \overline{r} \tag{A.7}$$

$$InputWaitData(r) = \overline{r} \tag{A.8}$$

$$NotifyInputTemporal(r) = \overline{r} \tag{A.9}$$

$$NotifyInputData(r) = \overline{r} \tag{A.10}$$

$$OutputWaitTemporal(r) = \overline{r} \tag{A.11}$$

$$OutputWaitData(r) = \overline{r} \tag{A.12}$$

$$NotifyOutputTemporal(r) = \overline{r} \tag{A.13}$$

$$NotifyOutputData(r) = (\nu\ f)\ \overline{part_y}.\overline{f} \mid f.\overline{r} \tag{A.14}$$

A.2 Process Purchase Order (ProcessPO)

$$ProcessPO\ = (\nu\ iwt,\ iwd,\ nit,\ nid,\ owt,\ owd,\ not,\ nod) \tag{A.15}$$

$$(iwt \mid iwd).(nit \mid nid).\tau_{interact}.(owt \mid owd).(not \mid nod) \tag{A.16}$$

$$\mid InputWaitTemp(iwt) \mid InputWaitData(iwd) \tag{A.17}$$

$$\mid NotifyInputTemporal(nit) \mid NotifyInputData(nid) \tag{A.18}$$

$$\mid OutputWaitTemporal(owt) \mid OutputWaitData(owd) \tag{A.19}$$

$$\mid NotifyOutputTemporal(nod) \mid NotifyOutputData(nod) \tag{A.20}$$

$$InputWaitTemporal(r) = \overline{r} \tag{A.21}$$

$$InputWaitData(r) = (\nu\ f)\ part_y.\overline{f}.(!part_y) \mid f.\overline{r} \tag{A.22}$$

$$NotifyInputTemporal(r) = \overline{r} \tag{A.23}$$

$$NotifyInputData(r) = \overline{r} \tag{A.24}$$

$$OutputWaitTemporal(r) = \overline{r} \tag{A.25}$$

$$OutputWaitData(r) = \overline{r} \tag{A.26}$$

$$NotifyOutputTemporal(r) = \overline{r} \tag{A.27}$$

$$NotifyOutputData(r) = (\nu\ f)(\overline{part_v}.\overline{f} \mid \overline{part_u}.\overline{f} \mid f.f.\overline{r})\ +\ (\nu\ f)(\overline{part_o}.\overline{f} \mid f.\overline{r}) \tag{A.28}$$

A.3 Request Financing Agreement

$$RFA = (\nu\ iwt,\ iwd,\ nit,\ nid,\ owt,\ owd,\ not,\ nod) \tag{A.29}$$

$$(iwt \mid iwd).(nit \mid nid).\tau_{interact}.(owt \mid owd).(not \mid nod) \tag{A.30}$$

$$\mid InputWaitTemp(iwt) \mid InputWaitData(iwd) \tag{A.31}$$

$$\mid NotifyInputTemporal(nit) \mid NotifyInputData(nid) \tag{A.32}$$

$$\mid OutputWaitTemporal(owt) \mid OutputWaitData(owd) \tag{A.33}$$

$$\mid NotifyOutputTemporal(nod) \mid NotifyOutputData(nod) \tag{A.34}$$

$$InputWaitTemporal(r) = \overline{r} \tag{A.35}$$

$$InputWaitData(r) = (\nu\ f)\ part_v.\overline{f}.(!part_v) \mid f.\overline{r} \tag{A.36}$$

$$NotifyInputTemporal(r) = (\nu\ f)\ \overline{temp_e}.\overline{f} \mid f.\overline{r} \tag{A.37}$$

$$NotifyInputData(r) = \overline{r} \tag{A.38}$$

$$OutputWaitTemporal(r) = \overline{r} \tag{A.39}$$

$$OutputWaitData(r) = \overline{r} \tag{A.40}$$

$$NotifyOutputTemporal(r) = \overline{r} \tag{A.41}$$

$$NotifyOutputData(r) = \overline{r} \tag{A.42}$$

A.4 Financing Agreement

$$FA = (\nu\ iwt,\ iwd,\ nit,\ nid,\ owt,\ owd,\ not,\ nod) \tag{A.43}$$

$$(iwt \mid iwd).(nit \mid nid).\tau_{interact}.(owt \mid owd).(not \mid nod) \tag{A.44}$$

$$\mid InputWaitTemp(iwt) \mid InputWaitData(iwd) \tag{A.45}$$

$$\mid NotifyInputTemporal(nit) \mid NotifyInputData(nid) \tag{A.46}$$

$$\mid OutputWaitTemporal(owt) \mid OutputWaitData(owd) \tag{A.47}$$

$$\mid NotifyOutputTemporal(nod) \mid NotifyOutputData(nod) \tag{A.48}$$

$$InputWaitTemporal(r) = \overline{r} \tag{A.49}$$

$$InputWaitData(r) = \overline{r} \tag{A.50}$$

$$NotifyInputTemporal(r) = \overline{r} \tag{A.51}$$

$$NotifyInputData(r) = \overline{r} \tag{A.52}$$

$$OutputWaitTemporal(r) = (\nu\ f)\ temp_e.\overline{f} \mid f.\overline{done} \mid done.(\overline{r} \mid !done) \tag{A.53}$$

$$OutputWaitData(r) = \overline{r} \tag{A.54}$$

$$NotifyOutputTemporal(r) = \overline{r} \tag{A.55}$$

$$NotifyOutputData(r) = (\nu\ f)\ \overline{part_r}.\overline{f} \mid f.\overline{r} \tag{A.56}$$

A.5 Request Shipping Agreement

$$RSA\ = (\nu\ iwt,\ iwd,\ nit,\ nid,\ owt,\ owd,\ not,\ nod) \tag{A.57}$$

$$(iwt \mid iwd).(nit \mid nid).\tau_{interact}.(owt \mid owd).(not \mid nod) \tag{A.58}$$

$$\mid InputWaitTemp(iwt) \mid InputWaitData(iwd) \tag{A.59}$$

$$\mid NotifyInputTemporal(nit) \mid NotifyInputData(nid) \tag{A.60}$$

$$\mid OutputWaitTemporal(owt) \mid OutputWaitData(owd) \tag{A.61}$$

$$\mid NotifyOutputTemporal(nod) \mid NotifyOutputData(nod) \tag{A.62}$$

$$InputWaitTemporal(r) = \overline{r} \tag{A.63}$$

$$InputWaitData(r) = (\nu\ f)\ part_u.\overline{f}.(!part_u) \mid f.\overline{r} \tag{A.64}$$

$$NotifyInputTemporal(r) = (\nu\ f)\ \overline{temp_f}.\overline{f} \mid f.\overline{r} \tag{A.65}$$

$$NotifyInputData(r) = \overline{r} \tag{A.66}$$

$$OutputWaitTemporal(r) = \overline{r} \tag{A.67}$$

$$OutputWaitData(r) = \overline{r} \tag{A.68}$$

$$NotifyOutputTemporal(r) = \overline{r} \tag{A.69}$$

$$NotifyOutputData(r) = \overline{r} \tag{A.70}$$

A.6 Shipping Agreement

$$SA = (\nu\ iwt,\ iwd,\ nit,\ nid,\ owt,\ owd,\ not,\ nod) \tag{A.71}$$
$$(iwt \mid iwd).(nit \mid nid).\tau_{interact}.(owt \mid owd).(not \mid nod) \tag{A.72}$$
$$\mid InputWaitTemp(iwt) \mid InputWaitData(iwd) \tag{A.73}$$
$$\mid NotifyInputTemporal(nit) \mid NotifyInputData(nid) \tag{A.74}$$
$$\mid OutputWaitTemporal(owt) \mid OutputWaitData(owd) \tag{A.75}$$
$$\mid NotifyOutputTemporal(nod) \mid NotifyOutputData(nod) \tag{A.76}$$

$$InputWaitTemporal(r) = \overline{r} \tag{A.77}$$
$$InputWaitData(r) = \overline{r} \tag{A.78}$$
$$NotifyInputTemporal(r) = \overline{r} \tag{A.79}$$
$$NotifyInputData(r) = \overline{r} \tag{A.80}$$
$$OutputWaitTemporal(r) = (\nu\ f)\ temp_f.\overline{f} \mid f.\overline{done} \mid done.(\overline{r} \mid !done) \tag{A.81}$$
$$OutputWaitData(r) = \overline{r} \tag{A.82}$$
$$NotifyOutputTemporal(r) = \overline{r} \tag{A.83}$$
$$NotifyOutputData(r) = (\nu\ f)\ \overline{part_q}.\overline{f} \mid f.\overline{r} \tag{A.84}$$

A.7 Analyse Finance Agreement and Shipping Agreement

$$AFASA = (\nu\ iwt,\ iwd,\ nit,\ nid,\ owt,\ owd,\ not,\ nod) \tag{A.85}$$
$$(iwt \mid iwd).(nit \mid nid).\tau_{interact}.(owt \mid owd).(not \mid nod) \tag{A.86}$$
$$\mid InputWaitTemp(iwt) \mid InputWaitData(iwd) \tag{A.87}$$
$$\mid NotifyInputTemporal(nit) \mid NotifyInputData(nid) \tag{A.88}$$
$$\mid OutputWaitTemporal(owt) \mid OutputWaitData(owd) \tag{A.89}$$
$$\mid NotifyOutputTemporal(nod) \mid NotifyOutputData(nod) \tag{A.90}$$

$$InputWaitTemporal(r) = \overline{r} \tag{A.91}$$

$$InputWaitData(r) = (\nu\ f)\ part_r.\overline{f}.(!part_r) \mid part_q.\overline{f}.(!part_q) \mid f.f.\overline{r} \tag{A.92}$$

$$NotifyInputTemporal(r) = \overline{r} \tag{A.93}$$

$$NotifyInputData(r) = \overline{r} \tag{A.94}$$

$$OutputWaitTemporal(r) = \overline{r} \tag{A.95}$$

$$OutputWaitData(r) = \overline{r} \tag{A.96}$$

$$NotifyOutputTemporal(r) = \overline{r} \tag{A.97}$$

$$NotifyOutputData(r) = (\nu\ f)\ \overline{part_o}.\overline{f} \mid f.\overline{r} \tag{A.98}$$

A.8 Purchase Order Confirmation (POC)

$$POC = (\nu\ iwt,\ iwd,\ nit,\ nid,\ owt,\ owd,\ not,\ nod) \tag{A.99}$$

$$(iwt \mid iwd).(nit \mid nid).\tau_{interact}.(owt \mid owd).(not \mid nod) \tag{A.100}$$

$$\mid InputWaitTemp(iwt) \mid InputWaitData(iwd) \tag{A.101}$$

$$\mid NotifyInputTemporal(nit) \mid NotifyInputData(nid) \tag{A.102}$$

$$\mid OutputWaitTemporal(owt) \mid OutputWaitData(owd) \tag{A.103}$$

$$\mid NotifyOutputTemporal(nod) \mid NotifyOutputData(nod) \tag{A.104}$$

$$InputWaitTemporal(r) = \overline{r} \tag{A.105}$$

$$InputWaitData(r) = (\nu\ f)\ part_o.\overline{f}.(!part_o) \mid f.\overline{r} \tag{A.106}$$

$$NotifyInputTemporal(r) = \overline{r} \tag{A.107}$$

$$NotifyInputData(r) = \overline{r} \tag{A.108}$$

$$OutputWaitTemporal(r) = \overline{r} \tag{A.109}$$

$$OutputWaitData(r) = \overline{r} \tag{A.110}$$

$$NotifyOutputTemporal(r) = \overline{r} \tag{A.111}$$

$$NotifyOutputData(r) = \overline{r} \tag{A.112}$$

A.9 Original

$$A = (\nu\ iwt,\ iwd,\ nit,\ nid,\ owt,\ owd,\ not,\ nod) \tag{A.113}$$

$$(iwt \mid iwd).(nit \mid nid).\tau_{interact}.(owt \mid owd).(not \mid nod) \tag{A.114}$$

$$\mid InputWaitTemp(iwt) \mid InputWaitData(iwd) \tag{A.115}$$

$$\mid NotifyInputTemporal(nit) \mid NotifyInputData(nid) \tag{A.116}$$

$$\mid OutputWaitTemporal(owt) \mid OutputWaitData(owd) \tag{A.117}$$

$$\mid NotifyOutputTemporal(nod) \mid NotifyOutputData(nod) \tag{A.118}$$

$$InputWaitTemporal(r) = (\nu\ f)\ \Pi_{i\in G}(temp_i.\overline{f}.(!temp_i)) \mid \{f\}_{j=|G|}.\overline{r} \tag{A.119}$$

$$InputWaitData(r) = (\nu\ f)\ \Pi_{i\in H}(part_i.\overline{f}.(!part_i)) \mid \{f\}_{j=|H|}.\overline{r} \tag{A.120}$$

$$NotifyInputTemporal(r) = (\nu\ f)\ \Pi_{i\in J}(\overline{temp_i}.\overline{f}) \mid \{f\}_{j=|J|}.\overline{r} \tag{A.121}$$

$$NotifyInputData(r) = (\nu\ f)\ \Pi_{i\in K}(\overline{part_i}.\overline{f}) \mid \{f\}_{j=|K|}.\overline{r} \tag{A.122}$$

$$OutputWaitTemporal(r) = \Pi_{i\in M}(\nu\ f)\big(\Pi_{j\in N}(temp_j.\overline{f}) \mid \tag{A.123}$$

$$\{f\}_{k=|N|}.\overline{done}\big) \mid \tag{A.124}$$

$$done.(\overline{r} \mid !done) \tag{A.125}$$

$$OutputWaitData(r) = \Pi_{i\in M}(\nu\ f)\big(\Pi_{j\in P}(part_j.\overline{f}) \mid \tag{A.126}$$

$$\{f\}_{k=|P|}.\overline{done}\big) \mid \tag{A.127}$$

$$done(\overline{r} \mid !done) \tag{A.128}$$

$$NotifyOutputTemporal(r) = \sum_{i\in M}\big((\nu\ f)\Pi_{j\in Q}(\overline{temp_j}.\overline{f}) \mid \{f\}_{k=|Q|}.\overline{r}\big) \tag{A.129}$$

$$NotifyOutputData(r) = \sum_{i\in M}\big((\nu\ f)\Pi_{j\in R}(\overline{part_j}.\overline{f}) \mid \{f\}_{k=|R|}.\overline{r}\big) \tag{A.130}$$

where

$$G = \{x \in TDG,\ y \in TD \mid y \in x \wedge sink(y) = A \wedge sinkType(y) = input\}$$

$$H = \{x \in DDG,\ y \in DD \mid y \in x \wedge sink(y) = A \wedge sinkType(y) = input\}$$

$$J = \{x \in TDG,\ y \in TD \mid y \in x \wedge source(y) = A \wedge sourceType(y) = input\}$$

$$K = \{x \in DDG,\ y \in DD \mid x \in y \wedge sourceTask(y) = A \wedge sourceType(y) = input\}$$

$$M = \{x \in DS \mid type(x) = output \wedge task(x) = A\}$$

$$N = \{x \in TDG,\ y \in TD \mid y \in x \wedge sinkgroup(y) = m\}$$

$$P = \{x \in DDG, y \in DD \mid y \in x \wedge sinkgroup(y) = m\}$$

$$Q = \{x \in TDG, y \in TD \mid y \in x \wedge sourcegroup(y) = m\}$$

$$R = \{x \in DDG, y \in DD \mid y \in x \wedge sourcegroup(y) = m\}$$

Appendix B

SSDL for the VO Example

B.1 Buyer

```
<?xml version="1.0" encoding="utf-8" ?>
<ssdl:contract targetNamespace="http://example.org/buyer" xmlns:ssdl="urn:ssdl:v1"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xsi:schemaLocation="urn:ssdl:v1:protocol:sc
               file:////Users/nsjw7/dev/SSDL/src/xsd/ssdl-protocol-seq-con.xsd
               urn:ssdl:v1 file:///Users/nsjw7/dev/SSDL/src/xsd/ssdl.xsd">
  <ssdl:schemas>
    <!-- schemas -->
  </ssdl:schemas>
  <ssdl:messages targetNamespace="http://example.org/buyer/messages"
  xmlns:formats="http://exaxmple.org/buyer/formats">
    <ssdl:message name="RequestPriceAvailability">
      <ssdl:body ref="formats:rpa"/>
    </ssdl:message>
    <ssdl:message name="PriceAvailability">
      <ssdl:body ref="formats:pa"/>
    </ssdl:message>
    <ssdl:message name="RequestPurchaseOrder">
      <ssdl:body ref="formats:rpo"/>
    </ssdl:message>
    <ssdl:message name="PurchaseOrder">
      <ssdl:body ref="formats:po"/>
    </ssdl:message>
    <ssdl:message name="ShippingStatus">
      <ssdl:body ref="formats:ss"/>
    </ssdl:message>
  </ssdl:messages>
  <ssdl:protocols>
    <ssdl:protocol targetNamespace="http://example.org/buyer/protocol"
    xmlns:msgs="http://example.org/buyer/messages"
                   xmlns:sc="urn:ssdl:v1:protocol:sc">
      <sc:sc>
        <sc:participant name="Seller"/>
        <sc:participant name="Delivery"/>
        <sc:protocol name="buyerProtocol">
          <sc:sequence>
            <ssdl:msgref ref="RequestPriceAvailabiltiy" direction="out"
            sc:participant="Seller"/>
            <ssdl:msgref ref="PriceAvailability" direction="in"
```

```
            sc:partipant="Seller"/>
            <sc:choice>
              <sc:sequence>
                <ssdl:msgref ref="RequestPurchaseOrder" direction="out"
                sc:participant="Seller"/>
                <ssdl:msgref ref="PurchaseOrder" direction="in"
                sc:partipant="Seller"/>
                <sc:choice>
                  <ssdl:msgref ref="ShippingStatus" direction="in"
                  sc:participant="Delivery"/>
                  <sc:nothing/>
                </sc:choice>
              </sc:sequence>
              <sc:nothing/>
            </sc:choice>
          </sc:sequence>
        </sc:protocol>
      </sc:sc>
    </ssdl:protocol>
  </ssdl:protocols>
</ssdl:contract>
```

B.2 Seller

```
<?xml version="1.0" encoding="utf-8" ?>
<ssdl:contract targetNamespace="http://example.org/seller" xmlns:ssdl="urn:ssdl:v1"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xsi:schemaLocation="urn:ssdl:v1:protocol:sc
               file:////Users/nsjw7/dev/SSDL/src/xsd/ssdl-protocol-seq-con.xsd urn:ssdl:v1
               file:///Users/nsjw7/dev/SSDL/src/xsd/ssdl.xsd">
  <ssdl:schemas>
    <!-- schemas -->
  </ssdl:schemas>
  <ssdl:messages targetNamespace="http://example.org/seller/messages"
                 xmlns:formats="http://exaxmple.org/seller/formats">
    <ssdl:message name="RequestPriceAvailability">
      <ssdl:body ref="formats:rpa"/>
    </ssdl:message>
    <ssdl:message name="PriceAvailability">
      <ssdl:body ref="formats:pa"/>
    </ssdl:message>
    <ssdl:message name="RequestPurchaseOrder">
      <ssdl:body ref="formats:rpo"/>
    </ssdl:message>
    <ssdl:message name="PurchaseOrder">
      <ssdl:body ref="formats:po"/>
    </ssdl:message>
    <ssdl:message name="RequestFinancingApproval">
      <ssdl:body ref="formats:rfa"/>
    </ssdl:message>
    <ssdl:message name="FinancingApproval">
      <ssdl:body ref="formats:fa"/>
    </ssdl:message>
    <ssdl:message name="RequestShippingOrder">
      <ssdl:body ref="formats:rso"/>
    </ssdl:message>
```

```
    <ssdl:message name="ShippingOrder">
      <ssdl:body ref="formats:so"/>
    </ssdl:message>
  </ssdl:messages>
  <ssdl:protocols>
   <ssdl:protocol targetNamespace="http://example.org/seller/protocol"
   xmlns:msgs="http://example.org/seller/messages"
    xmlns:sc="urn:ssdl:v1:protocol:sc">
     <sc:sc>
       <sc:participant name="Buyer"/>
       <sc:participant name="Delivery"/>
       <sc:participant name="Finance"/>
       <sc:protocol name="sellerProtocol">
         <sc:sequence>
           <ssdl:msgref ref="RequestPriceAvailabiltiy" direction="in"
           sc:participant="Buyer"/>
           <ssdl:msgref ref="PriceAvailability" direction="out"
           sc:partipant="Buyer"/>
           <sc:choice>
             <sc:sequence>
               <ssdl:msgref ref="RequestPurchaseOrder" direction="in"
               sc:participant="Buyer"/>
               <sc:choice>
                 <ssdl:msgref ref="PurchaseOrderConfirmation" direction="out"
                 sc:partipant="Buyer"/>
                 <sc:sequence>
                   <sc:parallel>
                     <sc:sequence>
                       <ssdl:msgref ref="RequestFinancingApproval" direction="out"
                       sc:participant="Finance"/>
                       <ssdl:msgref ref="FinancingApproval" direction="in"
                       sc:participant="Finance"/>
                     </sc:sequence>
                     <sc:sequence>
                       <ssdl:msgref ref="RequestShippingOrder" direction="out"
                       sc:participant="Delivery"/>
                       <ssdl:msgref ref="ShippingOrder" direction="in"
                       sc:participant="Delivery"/>
                     </sc:sequence>
                   </sc:parallel>
                   <ssdl:msgref ref="PurchaseOrderConfirmation" direction="out"
                   sc:partipant="Buyer"/>
                 </sc:sequence>
               </sc:choice>
             </sc:sequence>
             <sc:nothing/>
           </sc:choice>
         </sc:sequence>
       </sc:protocol>
     </sc:sc>
   </ssdl:protocol>
  </ssdl:protocols>
  </ssdl:contract>
```

B.3 Finance

```
<?xml version="1.0" encoding="utf-8" ?>
<ssdl:contract targetNamespace="http://example.org/finance" xmlns:ssdl="urn:ssdl:v1"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xsi:schemaLocation="urn:ssdl:v1:protocol:sc f
               ile:////Users/nsjw7/dev/SSDL/src/xsd/ssdl-protocol-seq-con.xsd
               urn:ssdl:v1 file:///Users/nsjw7/dev/SSDL/src/xsd/ssdl.xsd">
  <ssdl:schemas>
    <!-- schemas -->
  </ssdl:schemas>
  <ssdl:messages targetNamespace="http://example.org/finance/messages"
                 xmlns:formats="http://exaxmple.org/finance/formats">
    <ssdl:message name="RequestFinanceApproval">
      <ssdl:body ref="formats:rfa"/>
    </ssdl:message>
    <ssdl:message name="FinanceApproval">
      <ssdl:body ref="formats:fa"/>
    </ssdl:message>
  </ssdl:messages>
  <ssdl:protocols>
    <ssdl:protocol targetNamespace="http://example.org/finance/protocol"
                   xmlns:msgs="http://example.org/finance/messages"
                   xmlns:sc="urn:ssdl:v1:protocol:sc">
      <sc:sc>
        <sc:participant name="Seller"/>
        <sc:protocol name="financeProtocol">
          <sc:sequence>
            <ssdl:msgref ref="RequestFinanceApproval" direction="in"
            sc:participant="Seller"/>
            <ssdl:msgref ref="FinanceApproval" direction="out"
            sc:participant="Seller"/>
          </sc:sequence>
        </sc:protocol>
      </sc:sc>
    </ssdl:protocol>
  </ssdl:protocols>
</ssdl:contract>
```

B.4 Delivery

```
<?xml version="1.0" encoding="utf-8" ?>
<ssdl:contract targetNamespace="http://example.org/delivery" xmlns:ssdl="urn:ssdl:v1"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xsi:schemaLocation="urn:ssdl:v1:protocol:sc
                file:////Users/nsjw7/dev/SSDL/src/xsd/ssdl-protocol-seq-con.xsd
                urn:ssdl:v1 file:///Users/nsjw7/dev/SSDL/src/xsd/ssdl.xsd">
  <ssdl:schemas>
    <!-- schemas -->
  </ssdl:schemas>

  <ssdl:messages targetNamespace="http://example.org/delivery/messages"
                 xmlns:formats="http://exaxmple.org/delivery/formats">
    <ssdl:message name="RequestShippingOrder">
      <ssdl:body ref="formats:rso"/>
    </ssdl:message>
```

```
16      <ssdl:message name="ShippingOrder">
17        <ssdl:body ref="formats:sa"/>
18      </ssdl:message>
19      <ssdl:message name="ShippingStatus">
20        <ssdl:body ref="formats:ss"/>
21      </ssdl:message>
22    </ssdl:messages>
23
24  <ssdl:protocols>
25      <ssdl:protocol targetNamespace="http://example.org/delivery/protocol"
26                     xmlns:msgs="http://example.org/delivery/messages"
27                     xmlns:sc="urn:ssdl:v1:protocol:sc">
28        <sc:sc>
29          <sc:participant name="Seller"/>
30          <sc:participant name="Buyer" abstract="true"/>
31          <sc:protocol name="deliveryProtocol">
32            <sc:sequence>
33              <ssdl:msgref ref="RequestShippingOrder" direction="in"
34              sc:participant="Seller"
35                           sc:participant-binding-name="Buyer"
36                           sc:participant-binding-content=
37                           "/soap:envelope/soap:header/buyer/wsa:EndpointReference/"/>
38              <ssdl:msgref ref="ShippingOrder" direction="out"
39              sc:participant="Seller"/>
40              <sc:choice>
41                <sc:nothing/>
42                <ssdl:msgref ref="ShippingStatus" direction="out"
43                sc:partipant="Buyer"/>
44              </sc:choice>
45            </sc:sequence>
46          </sc:protocol>
47        </sc:sc>
48      </ssdl:protocol>
49    </ssdl:protocols>
50    </ssdl:contract>
```

Appendix C

π-calculus representation of the SSDL SC Example

$$Buyer(buyer, seller) = \overline{seller} < rpa > .buyer(pa). \tag{C.1}$$

$$(\overline{seller} < rpo > .buyer(po). \tag{C.2}$$

$$(buyer(ss) + 0) \tag{C.3}$$

$$+ 0) \tag{C.4}$$

$$Seller(seller, buyer, delivery, finance) = (\nu\ d, f) \tag{C.5}$$

$$(seller(rpa).\overline{buyer} < pa > . \tag{C.6}$$

$$(seller(rpo).\overline{d} < r > .\overline{f} < r > .r.r.\overline{buyer} < poc > \tag{C.7}$$

$$+\ 0)) \tag{C.8}$$

$$|\ d(comp).\overline{delivery} < buyer, rso > .[x = so]seller(x).\overline{comp} \tag{C.9}$$

$$|\ f(comp).\overline{finance} < rfa > .[x = fa]seller(x).\overline{comp} \tag{C.10}$$

$$\tag{C.11}$$

$$Delivery(delivery, seller) = \tag{C.12}$$

$$delivery(buyer, rso)\overline{seller} < rso > .\overline{buyer} < ss > \tag{C.13}$$

$$Finance(finance, seller) = \tag{C.14}$$

$$finance(rfa).\overline{seller} < fa > \tag{C.15}$$

$$Interaction = (\nu\ buyer, seller, delivery, finance) \tag{C.16}$$

$$Buyer(buyer, seller) \mid Seller(seller, buyer, delivery, finance) \mid \tag{C.17}$$

$$Delivery(delivery, seller) \mid Finance(finance, seller) \tag{C.18}$$

Appendix D

XML Schema for the Task Model

```
<?xml version="1.0"?>
<xsd:schema targetNamespace="http://schemas.adapt.org/process-definition-2.1/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:tns="http://schemas.adapt.org/process-definition-2.1/"
elementFormDefault="qualified" attributeFormDefault="unqualified">
        <xsd:import namespace="http://schemas.xmlsoap.org/wsdl/"
        schemaLocation="wsdl.xsd"/>
        <xsd:element name="processDefinition" type="tns:ProcessDefinitionRootType"/>
        <xsd:complexType name="ImportType">
                <xsd:attribute name="namespace" type="xsd:anyURI" use="required"/>
                <xsd:attribute name="location" type="xsd:anyURI" use="required"/>
        </xsd:complexType>
        <xsd:complexType name="ProcessDefinitionRootType">
                <xsd:sequence>
                        <xsd:element name="import" type="tns:ImportType" minOccurs="0"
                        maxOccurs="unbounded"/>
                        <xsd:element name="subProcesses" type="tns:SubProcessesType"
                        minOccurs="0"/>
                        <xsd:element name="outputDependencies"
                         type="tns:DependencyListType"
                        minOccurs="0"/>
                </xsd:sequence>
                <xsd:attributeGroup ref="tns:ActionAttributes"/>
                <xsd:attribute name="targetNamespace" type="xsd:anyURI"/>
        </xsd:complexType>
        <!-- Elements to describe the process and its structure -->
        <xsd:complexType name="ProcessDefinitionType">
                <xsd:choice>
                        <xsd:sequence>
                                <xsd:element name="inputDependencies"
                                type="tns:DependencyListType"
                                minOccurs="0"/>
                                <xsd:element name="subProcesses"
                                type="tns:SubProcessesType"
                                minOccurs="0"/>
                                <xsd:element name="outputDependencies"
                                type="tns:DependencyListType"
                                minOccurs="0"/>
                        </xsd:sequence>
                </xsd:choice>
                <xsd:attributeGroup ref="tns:ActionAttributes"/>
                <xsd:attribute name="definition" type="xsd:QName"/>
```

```
        </xsd:complexType>
        <xsd:complexType name="TaskDefinitionType">
                <xsd:choice>
                        <xsd:element name="inputDependencies"
                         type="tns:DependencyListType"
                        minOccurs="0"/>
                </xsd:choice>
                <xsd:attributeGroup ref="tns:ActionAttributes"/>
                <xsd:attribute name="port" type="xsd:QName"/>
        </xsd:complexType>
        <xsd:complexType name="SubProcessesType">
                <xsd:sequence>
                        <xsd:choice maxOccurs="unbounded">
                                <xsd:element name="taskDefinition"
                                type="tns:TaskDefinitionType"/>
                                <xsd:element name="processDefinition"
                                type="tns:ProcessDefinitionType"/>
                        </xsd:choice>
                </xsd:sequence>
        </xsd:complexType>
        <!-- Elements to describe the dependencies between tasks and processes -->
        <xsd:complexType name="DependencyType">
                <xsd:attributeGroup ref="tns:DependencyAttributes"/>
        </xsd:complexType>
        <xsd:complexType name="DataDependencyType">
                <xsd:attributeGroup ref="tns:DependencyAttributes"/>
                <xsd:attribute name="sourcePartName" type="xsd:string" use="optional"/>
                <xsd:attribute name="sinkPartName" type="xsd:string" use="optional"/>
        </xsd:complexType>
        <xsd:complexType name="DependencyListType">
                <xsd:sequence>
                        <xsd:element name="dependency" type="tns:DependencyType"
                        minOccurs="0"
                        maxOccurs="unbounded"/>
                        <xsd:element name="dataDependency" type="tns:DataDependencyType"
                         minOccurs="0"
                        maxOccurs="unbounded"/>
                </xsd:sequence>
        </xsd:complexType>
        <!-- Utility definitions to define common attributes of tasks/processes and
        dependencies -->
        <xsd:attributeGroup name="ActionAttributes">
                <xsd:attribute name="name" type="xsd:QName" use="required"/>
                <xsd:attribute name="portType" type="xsd:QName" use="optional"/>
                <xsd:attribute name="operation" type="xsd:NCName" use="optional"/>
                <xsd:attribute name="lateInstantiation" type="xsd:boolean" use="optional"
                default="false"/>
                <xsd:attribute name="logicalHost" type="xsd:QName" use="required"/>
                <xsd:attribute name="logicalService" type="xsd:QName" use="optional"/>
        </xsd:attributeGroup>
        <xsd:attributeGroup name="DependencyAttributes">
                <xsd:attribute name="source" type="xsd:QName" use="required"/>
                <xsd:attribute name="sourceMessageType" type="tns:MessageTypeType"
                use="required"/>
                <xsd:attribute name="sourceMessageName" type="xsd:string" use="optional"/>
                <xsd:attribute name="sinkMessageType" type="tns:MessageTypeType"
                use="optional"/>
                <xsd:attribute name="sinkMessageName" type="xsd:string" use="optional"/>
```

```
101                 <xsd:attribute name="priority" type="xsd:integer" use="optional"
102                 default="0"/>
103         </xsd:attributeGroup>
104         <xsd:simpleType name="MessageTypeType">
105                 <xsd:restriction base="xsd:string">
106                         <xsd:enumeration value="input"/>
107                         <xsd:enumeration value="output"/>
108                         <xsd:enumeration value="fault"/>
109                 </xsd:restriction>
110         </xsd:simpleType>
111 </xsd:schema>
```

Appendix E

Connected Workflow for All Parties

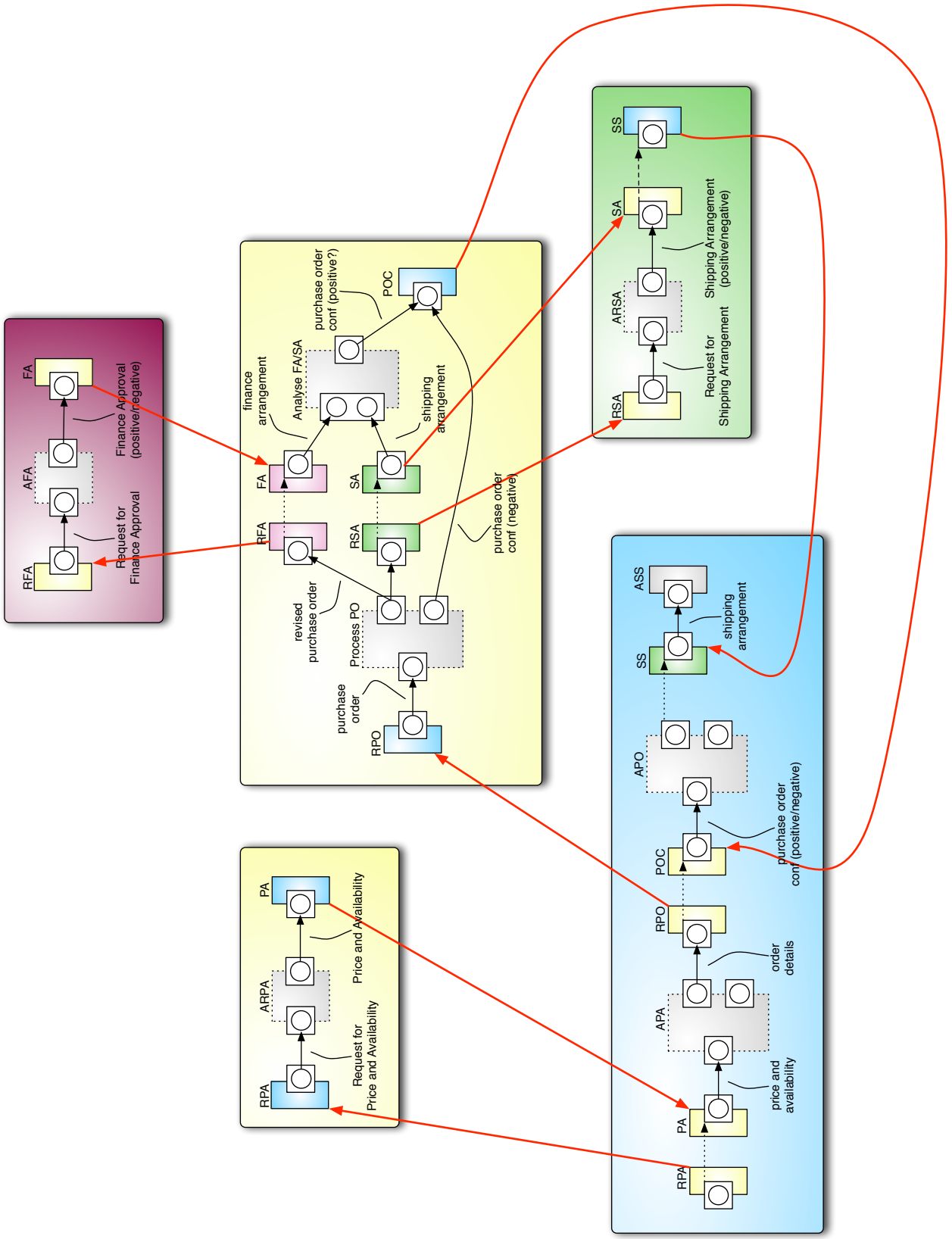


Figure E.1: Complete Task Model Example