

Adaptive service-oriented systems: an organisational approach

Alan Colman and Jun Han

*Faculty of Information and Communication Technologies, Swinburne University of Technology, Melbourne, Victoria, Australia
Email: {acolman,jhan}@swin.edu.au*

The relationships between the loosely coupled services of a composition can be non-deterministic and unreliable. This requires that a composite application has the ability to adaptively reconfigure itself to continuously satisfy the system requirements. Imperative composition approaches such as BPEL do not provide the abstractions necessary to create adaptive compositions. In this paper we introduce an organisation-oriented, application-centric service composition framework that aims to achieve adaptive application behaviour from non-deterministic and unreliable services. The responsibility for coordinating and managing service interactions resides with a coordination/management layer of the application itself. In this layer, contracts define and regulate the relationships between the service roles in a composition. By creating and revoking contracts between roles and by binding service roles to individual services, a composition can be adaptively reconfigured in response to changing requirements and conditions.

Keywords: Service composition, roles, contracts, organisation

1. INTRODUCTION

Web services can provide a means for integrating disparate applications/systems within a single organisation (EAI), or they can provide the middleware layer for the integration of systems in different organisations (B2B). In the above cases, Web services expose external functions of systems to be integrated, and provide the mechanisms for their interaction. From this perspective, Web services are viewed as a means for integrating systems that are peers. An alternative architectural perspective on Web services is to see them as providers of needed functions for an application. From this perspective, an application uses Web services to fulfil requirements it cannot provide for itself. In this type of system, the application is a management layer that coordinates

the functionality provided by Web services or other components. Rather than being just a process 'glue' that holds external Web services together, this management layer is structured. It is this architectural view – a central application with loosely coupled satellite services – that we are focusing on in this paper.

In order to compose Web services with other services or applications, the services need to be functionally compatible, the interactions between the services need to be well-ordered, and the composed behaviour must meet any requirements for the system as a whole. However, in the open and distributed environment of the Internet, the relationships between loosely coupled services can be non-deterministic. In an application-centred architecture, the central application needs to be able to meet its functional and non-functional

requirements, even if the Web services it relies upon are to some degree unpredictable. In particular, if an application needs to guarantee a certain level of performance, and yet relies on distributed Web services to provide some functionality, the application needs a mechanism to manage the quality of service (QoS) of its subsidiary services. For example, if the application is a work flow scheduling system, the system needs to respond dynamically to changes in demand, and to variations in the performance of its constituent services. It must monitor and restructure the interactions of the services as loads on those services change. As such, it requires the composite system to have management capabilities tailored to the application domain.

Much work has been done on the preparation of standards to allow the development of Web service compositions that are well-behaved and meet the requirements of the composite system. While the point-to-point simple interactions have reached a level of maturity in their standardisation and implementation, achieving reliable behaviour from more complex configurations of Web services is still an open problem. The development of standards to address such behavioural and non-functional aspects of complex composed services is being undertaken on a number of fronts: for example, coordination (BEA Systems *et al.* 2004), choreography (W3C, 2004), orchestration (BEA Systems *et al.* 2003), and management (OASIS, 2005). Similar issues have been addressed in the Grid service domain (Global Grid Forum, 2004).

Many of these standardisation efforts define a management or coordination level of abstraction. In these standards, management and coordination functions are encapsulated as services with a well-defined Web service interface. Such an approach may accord well with the philosophy of making all functions accessible through a WSDL (W3C, 2005) interface. However, having heavy-weight middleware that provides encapsulated management and coordination services may not be the best way to achieve good behaviour and required performance in the application-centred architectures described above. It is unclear how the programmer can use these various overlapping and sometimes incompatible middleware standards to achieve a well regulated, adaptive application. While the composition language BPEL (BEA Systems *et al.* 2003) facilitates the composition of Web services into an application, functional composition and adaptive management concerns get mixed into a single process description. What is needed is a framework that provides the necessary adaptive management and coordination functions that the application programmer can readily extend to suit domain and application specific requirements. In this paper, we propose such a framework – the Role Oriented Adaptive Design (ROAD) framework for Web services. Like a number of WS standards, the ROAD framework defines a separate coordination/management layer. However, in ROAD this layer is superimposed on the pre-existing decoupled application components and services, and creates an organisational structure for those services, rather than being a separate encapsulated service provided by the middleware. The primary responsibility for coordination and management of the interactions resides with this coordination/management layer of the application.

The structure of the paper is as follows. In Section 2 we define the concept of an adaptive system in terms of

managed indirection of instantiation and composition. Section 3 outlines a role-based architectural framework that supports managed indirection to achieve adaptivity. In this framework functional roles can be played by Web services. Roles are dynamically associated by contracts. Contracts can be defined at various levels of abstraction. They provide interaction control and performance monitoring. An organiser creates and manages contracts between roles within a self-managed composite. It also binds roles to concrete services. The organiser changes contracts and bindings within its composite to achieve adaptivity. Section 4 briefly discusses the implementation of the ROAD framework and a test application. Section 5 discusses related work and Section 6 concludes and discusses further work.

2. ADAPTIVE SYSTEMS – THE MANAGEMENT OF INDIRECTION

We define an adaptive software system as one that can reconfigure itself in response to changes in its environment, or in response to changes in its own goals or capabilities. The environment of a software application can include the Web services that it relies on to function and are outside its scope of control. These services may have variable reliability, availability, performance etc. Even if an application's services have such unpredictable QoS (quality of service), we still need to achieve some acceptable level of system-wide performance. An application needs to be able to recognise when one of its services is underperforming, and then take remedial action. Such action might include reorganising the work load, reconfiguring the interactions between services, or replacing the underperforming service. Adaptable applications are those that can be readily recomposed. To achieve recomposition, the system structure must be flexible with controlled indirection between the entities in the system.

2.1 Indirection is required for adaptivity

Indirection in Web service compositions can be created in two dimensions. We will call these two dimensions indirection of instantiation, and indirection of composition. Figure 1 illustrates these two types of indirection that give software systems the flexibility needed to be adaptive. Firstly, the abstract description of services can be distinguished from

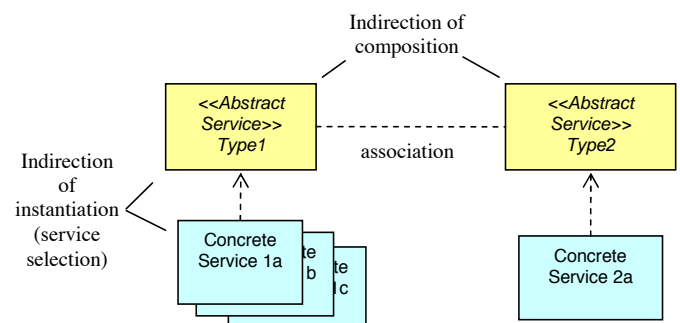


Figure 1 Two dimensions of indirection in Web services

their concrete implementation (as in the separation of the abstract and concrete parts of a WSDL specification (W3C, 2005)). In an adaptive system, such binding should be able to be created and destroyed at runtime. Such dynamic binding of actual services to abstract service-types is well supported in Web services (in specification at least) through service discovery and selection mechanisms.

The second type of indirection is found in the association between the abstract services. Associations between service types might be hard-coded references in the implementation of a client service, or, perhaps more typically, scripted using an imperative composition language such as BPEL (BEA Systems *et al.*, 2003). Such associations are abstracted by using reference variables, or by explicitly representing an association type (such as a partner-link type as in BPEL). However, in the absence of other mechanisms, all possible runtime determinations (resolutions) of the association indirection must be anticipated at design time (e.g. in BPEL as cases in a switch structure). The indirection in the association is not per se adaptive because the associations are hard-coded when the service or the composition script is implemented.

While the mechanism of runtime determination of instantiation is supported in Web services, the determination of composition between service types is often fixed during implementation thereby limiting the adaptivity of the application.

2.2 Indirection must be managed to achieve adaptivity

In a functioning system any indirection must be determined before or during runtime. An adaptive composite system needs to manage its indirection; that is, it needs to be able to dynamically create bindings between its loosely coupled elements on both the above dimensions in response to changing demands and changing environment. To do this, the adaptive system needs to be able to perform the following management and binding functions:

- monitor the performance of its current configuration
- have access to alternative services (with various performance characteristics) and have the ability to select between services
- ensure associated services are functionally compatible and have compatible interaction sequences
- check the validity and (optionally) evaluate the performance of alternative compositions
- reconfigure its bindings and associations.

The basic Web service standards of WSDL and UDDI partially address some of these functions (basic functional compatibility and service selection respectively). There has been a recent proliferation of standards that address some of the other functions listed above. WS-Coordination (BEA Systems *et al.* 2004) and WS-CDL (W3C, 2004) address the compatibility of interaction protocols, Management of Web Services (MoWS) (OASIS, 2005) and WS-Agreement (Global Grid Forum, 2004) address monitoring and performance issues, and so on. These standards are to some extent overlapping and incompatible, and they do not provide clear

guidance on how to integrate and use such standards if we want to develop an adaptive application as characterised above. For example, BPEL allows us to define compositions from services, but does not support performance monitoring in its orchestration model other than fault handling and compensation. If the programmer wants to monitor the performance of a service, the monitoring code is tangled with the orchestration script.

As well as providing the necessary indirection, an adaptive architecture must show what elements in the system are responsible for the binding and management of that indirection, and how these elements can achieve adaptive adjustment.

3. AN ADAPTIVE FRAMEWORK FOR USING WEB SERVICES

In this section we introduce a framework to facilitate the development of adaptive applications built from Web services. We call this framework Role Oriented Adaptive Design (ROAD). The design aims of this framework are as follows:

- facilitate the development of adaptive composite applications that can perform reliably while using Web services whose performance is unreliable
- provide a library of constructs that can be extended by the programmer to compose, coordinate and manage adaptive composites
- allow the management/coordination layer to be coded as a separate concern from the functional roles and services
- have compositions that can be imposed on services without the services needing to be (necessarily) written to horizontal standards such as WS-Coordination.

This section outlines the basic elements of the ROAD framework, and shows how it can be used to create adaptive compositions that manage indirection of instantiation and composition.

3.1 Basic concepts

The adaptive framework for using Web services introduced here is based on the conceptual separation of roles from the entities that play those roles, and the associations of roles with contracts. Applications are viewed as organisations –

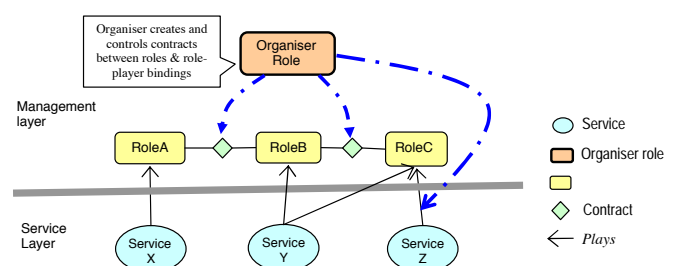


Figure 2 Roles, contracts and organisers from a management layer separate from services

goal-driven networks of roles bound together by contracts. Roles can be played by various players (services) in much the same way as a role in a business structure may be played by various employees, or outsourced to external organisations. Similarly, roles in the adaptive service-oriented application can be played by Web services or other components that are within the organisation, or by external Web services outside the application's immediate scope of control. Players can be dynamically bound/unbound (indirection of instantiation) to roles as demands on the application changes, or as the players' performance varies.

As shown in Figure 2, ROAD contracts associate functional roles. They also monitor and regulate interactions between the roles. As all roles (as opposed to the services) are internal to the organisation, contracts are also internal (although these internal contracts may be mirrored in external SLAs). Contracts define the mutual obligations of the participant roles in an organisational context. They define what interactions are permissible or required by the participant roles, and can be used to enforce sequences of interactions (conversations). Contracts can also be used to set performance conditions on the roles' interactions, and monitor those interactions for compliance to those conditions (performance management). ROAD contracts thus encapsulate three functions: composition, interaction control, and performance management.

Organisers make and break the bindings between functional roles and players (service selection), and create and revoke the contracts between the roles. They can thereby create various configurations of roles and services. Organisers set performance requirements for the contracts they control, and receive performance information from those contracts. Organisers are themselves a role-player pair, so that role-players of varying capability can be bound to the organiser role. In short, organisers provide the adaptivity to the application by managing the indirection of composition and instantiation. Organisers are responsible for the configuration of a set of roles and contracts. We call such configurations self-managed composites.

Figure 2 illustrates the relationship between these concepts. Organisers, along with the contracts and roles that they control, can be viewed as a management layer that composes and controls the interaction of the underlying services. In this way the ROAD architecture is not a client-server in the traditional sense. The fundamental difference is the distinction between management and functional layers. The essential ROAD application is an instantiated structure of roles (service definitions), contracts and organisers that creates a management system for the underlying services. All messages between services pass through the roles and the contracts that associate them. However, all functional processing is done by role-players, which can either be inside or outside the organisational boundary. In some cases, all players in an application may be external services, such that there is no functional processing carried out within the organisational boundary.

In this sense, the ROAD framework has more in common with a compositional approach, such as BPEL, than with a traditional client-server architecture: services are selected to play (instantiate) the roles (service types) in the application organisation (service composition). The difference with BPEL is that ROAD is based on dynamically contracting roles (achieved by the additional management layer) rather than procedurally linking activities. The following subsections explain these concepts in more detail.

3.2 Roles and players

Functional roles are runtime application entities that hold abstract service definitions. Because the services that play roles can be transitory (for example, there may be no service currently available to play a role), roles also maintain any state (e.g. message queues) associated with its position in the organisation. To illustrate the concept of a role, consider a highly simplified production management system application for a Widget manufacturing department. Functional work units within the department are represented as functional roles. These roles can be played by heterogenous types of role-player, including services that are external to the organisational boundary or services/components that are internal to the organisation. Figure 3 illustrates four roles in our Widget department: a ThingyMaker makes thingies, a DooverMaker makes doovers, an Assembler assembles thingies and doovers into widgets, and a Foreman allocates work to the other players. In our example, thingies and doovers are obtained from external suppliers (Services X, Y, Z) and made into Widgets by an Assembler (p1 is an internal player; an interactive UI that require human employees who do the physical assembly to record their work). The Foreman is played by legacy scheduling software wrapped in a Web service interface (p2). Roles can be thought of as a proxies within the organisation for services irrespective of whether those services are internal or external. The role includes a WSDL abstract service definition that is made concrete with a service endpoint reference when a service is bound to the role. Service selection (e.g. between ServiceX and ServiceY) is not the responsibility of the role itself, but the responsibility of the composite's organiser (as discussed below).

3.3 Contracts

As pointed out above, ROAD contracts perform three functions in an organisational structure: composition, interaction control, and performance management. ROAD contracts are dynamic, rich associations between roles. By creating and/or revoking contracts, the topology of the composition of roles can be altered. For example, in Figure 3, suppose a role instance t1 of type ThingyMaker (played y ServiceY) is unable to meet the requirements as defined in its contract with Foreman f1. An alternative to replacing ServiceY with a more capable service (say ServiceX) would be to add another instance t2 of type ThingyMaker (played by ServiceZ) to the

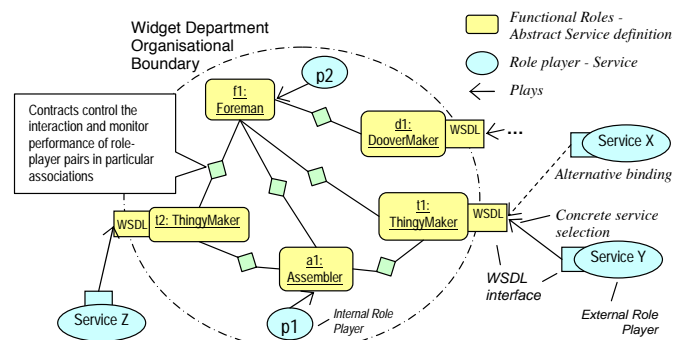


Figure 3 Creating flexible role-structures with role-player bindings and role-role contracts

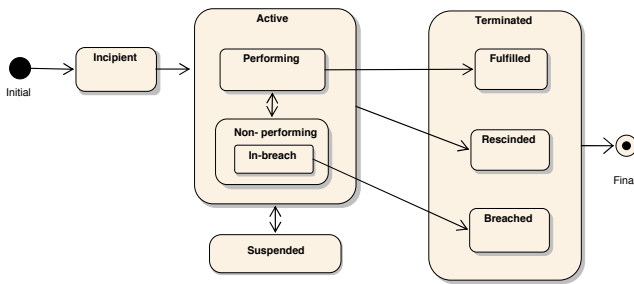


Figure 4 Life-cycle of a contract showing various states

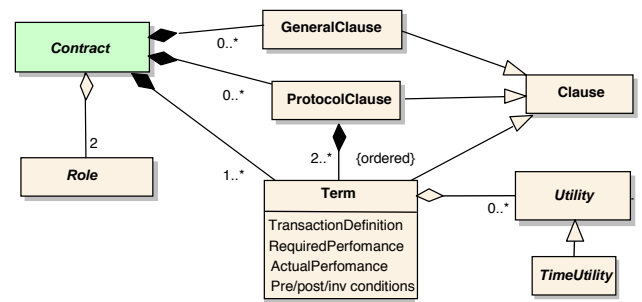


Figure 6 ROAD contracts - basic concepts

role structure. This changes the topology of the organisation.

Contracts, however, provide more than simple references between role types. They monitor and regulate interactions between the roles by intercepting the messages that pass between them. Contracts define the mutual obligations of the participant roles. They define what interactions are permissible or required by the participant roles, and can be used to enforce sequences of interactions. Contracts can also be used to set QoS conditions on the interactions (service level objectives), and monitor those interactions for compliance to those conditions.

3.3.1 A definition of contracts

ROAD contracts (Colman and Han, 2005b) are binary association classes that express the obligations of the contract parties to each other. They define the functional interactions that can occur between the role players. They also define the non-functional requirements of each of the parties with respect to those interactions, and measure the role-players' performances against those requirements. ROAD contracts can be characterised by the following features (illustrated in Figure 6):

- The names of the parties to the contract. A ROAD contract type binds roles of particular types (e.g. Foreman, ThingyMaker).

Contracts will also have a number of clauses. Clauses can be of three types: terms; general clauses; and protocol clauses:

- The *terms* of a contract are clauses that specify what one party can ask of the other party. The collection of terms

defines the parties' mutual functional obligations. For example, a contract term may specify that a ThingyMaker is obliged to fulfil requests to make thingies from its Foreman. Non-functional attributes (utilities) are associated with those terms – for example, the minimum performance standard, the price, quality of service etc. Each term of the contract can have one or more agreed utility functions that measure this performance. Contracts may also contain provisions that define remedies if a clause is breached, or if there is underperformance, by one of the parties. Some terms may “go to the heart of the contract” in which case breach of a clause leads to termination of the contract.

- *General clauses* in a contract define the preconditions for the contract's instantiation. These include any conditions relating to commencement, continuation, and termination of the contract.
- *Protocol clauses* define sequences of terms to be followed by the parties (Han and Ker, 2003; Plasil and Visnovsky, 2002). For example, a Foreman might be required to allocate the resources (thingy parts) to a ThingyMaker, before it can ask the ThingyMaker to make a thingy. A number of existing approaches exist for specifying interaction protocols (Bracciali *et al.*, 2002) and patterns (Ortiz *et al.*, 2004). While we will not discuss protocol clauses in this paper any further, we envisage extending the ROAD framework by using one such existing approach.

As well as having the above attributes, ROAD contracts have a number of incarnations: *general form* (*à la* class),

Contract Type Foreman-ThingyMaker Contract	
Contract Instance Name	ft1
Parties Party A: Foreman	f1
Party B: ThingyMaker	tm1
Terms	
1. B must make thingies on request from A PRE: qty (thingyparts) > 0	NFR1 Moving average >= 5 thingies / minute NFR2: cost = \$1 / thingy
2. A may provide thingy parts to B	NFR: none
Breach conditions	B provides < 2 thingies / minute
General Clauses	contract state is suspended if B under maintenance
Current state	incipient

Figure 5 Example of a contract instance between a Foreman and a ThingyMaker

specific contract (à la object) and an *execution state*.

- The *general form* (type) of a contract sets out the mutual obligations and interactions between parties of particular classes (e.g. Foreman, ThingyMaker). Clauses applicable to all contracts of that type can be defined. Such clauses may express interaction patterns (Party A will do X when Party B has done Y). Clauses may themselves be the subject of interaction patterns (Clause 2 will take effect after Clause 1 is fulfilled).
- A *specific contract* puts values against the variables in the contract *schedule* (e.g. Foreman and ThingyMaker are named, date of commencement agreed, performance conditions put on clauses etc.). Extra clauses, not in the general form of contract, may also be added. A specific contract is *instantiated* with an identity when the concrete contract is ‘signed’ – that is, when specific parties are bound to the contract.
- The terms of a contract have *execution states*. Each term has a state machine that maintains the state of interaction between the parties with respect to that term (e.g. Party A has asked Party B to do a under the terms of Term X, but Party B has not yet complied). We call the interaction that occurs during the execution of a term, a *transaction*. Contracts also have an execution state. States of an instantiated contract can include incipient, active, suspended, and terminated. Active and terminated states can have a number of sub-states as shown in the diagram of a contract life-cycle¹ in Figure 4.

The transitions between these contract states are initiated by the contract’s owner (the organiser of its self-managed composite), or by conditions in the clauses being triggered. The triggering of conditions in contract *terms* can cause transitions between the *active state* sub-states. For example, Figure 5 is an example of a (partial) Foreman-ThingyMaker Contract. The specific schedule values of the contract instance are in italics. If the contract state is “Performing”, and Term 1 is violated (the ThingyMaker makes less than 5 thingies per minute), then the contract state will transition to “Non-performing” or “Inbreach” (if the rate is less than 2 thingies per minute). If there are no general clauses that match this circumstance, the Organiser needs to make a decision as to whether it will transition the contract to one of the “terminated” states. This is not necessarily an automatic transition as the Organiser may have no other thingyMaker services available, and may choose to keep an underperforming player. Note that contract states do not explicitly indicate the “guilty party”, but this information is implicit in the term that is violated (Party B in the case of Term 1).

Figure 6 shows the basic concepts that make a ROAD contract, and that serve as a meta-model for the implementation of contracts in the ROAD framework. The elements in the diagram are discussed in more detail in subsequent sections. Real-world commercial contracts are passive bits of paper that are monitored, and to some extent enforced, by the parties. A ROAD software contract, on the other hand, can store dynamic state-of-execution information in the contract

itself. It can also enforce the terms of the contract by controlling the interactions between the parties.

3.3.2 Interaction pattern enforcement.

Contracts between functional roles often share common characteristics. In particular, the control aspects of the contract can be abstracted. Using our example from Figure 3, the control-management relationship between a Foreman and an Assembler could be characterised as a Supervisor-Subordinate relationship, while a ThingyMaker-Assembler relationship could be characterised a Supplier-Acquirer relationship. Operational-management contracts define the interaction protocols between operational-management roles such as a supervisor and a subordinate. In ROAD, these abstracted control aspects of the relationships between roles are encapsulated in abstract operational-management contracts (“management contracts” for short).

Management contracts rely on message abstraction to define coordination protocols, and they control interaction according to those protocols. Abstract messages characterise the direction and control aspect of a message (e.g. request, reply, invoke, set etc.). For example, a supervisor can request a subordinate to act and the subordinate must comply. On the other hand, a subordinate *cannot* invoke an action in a supervisor. A more detailed discussion of ROAD management contracts can be found in Colman & Han (2005b). Concrete contracts between roles (e.g. a Foreman-ThingyMaker contract) inherit interaction control patterns from such abstract management contracts. The protocols defined in management contracts are, to an extent, domain-independent as similar interaction patterns reoccur in many domains. They may be as general as the Message Exchange Patterns (MEPs) in WDSL (W3C, 2005), or they may be defined with richer semantics. Generality can facilitate the reuse of such abstract contracts. However, all roles (as opposed to the players) are internal to the organisation, and contracts themselves are internal to the application. Management contracts can therefore be defined or extended as the particular application requires. They do not need to rely on patterns of interaction predefined in external standards (although they may make use of them).

A state machine in the contract (Colman and Han, 2005a) can define and enforce a set of valid conversations between the parties to the contract. The contract at this level acts as a conversation controller (Alonso, Casati, Kuno and Machiraju, 2004). This mechanism can also be used to prevent communication between a party to a contract and an entity that is not bound by the contract.

3.3.3 Performance monitoring.

ROAD contracts specify the *required* performance of interactions between the parties, and they monitor and store *actual* performance. These contracts between the roles in the organisation can be thought of as service level agreements (SLAs) that monitor their own terms.

The monitoring of service performance using the ROAD framework is carried out by intercepting messages as they pass through the contracts that bind the services’ roles. This monitoring is extrinsic; that is, it is performance of a concrete role-player pair as measured from the application that is using the service. The application does not have to rely on abstract descriptions of *claimed* service performance

1. (Beugnard *et al.*, 1999) defines a similar scheme for contract life-cycle, namely: Definition (selection of contract type as suitable), subscription (parties bound to contract), application (including handling contract violations), termination (parties unbound), and deletion. In ROAD, on the other hand, parties are bound to a contract when it is instantiated.

- Make and break contracts between roles in its self-managed composite. An organiser can instantiate, change, abrogate and reassign contracts between functional roles.
- Make and break bindings between its roles and services (setting the service endpoint reference in the role).
- Monitor the actual performance of contracts in their self-managed composite. This can occur through the contract notifying its organiser of underperformance or breach. Alternatively, the organiser can poll its contracts.
- Update the required performance in the terms of a contract.
- Change the conditions in both general clauses and terms of a contract.
- Receive non-functional requirements from organisers higher up the coordination system
- Transmit non-functional requirements to the organisers of any sub-composites players under its control

In order to carry out these operations an organiser must maintain a representation of the organisation. It must know:

- What roles and services it is controlling.
- What contract-types it has available to associate those roles.
- What potential services are available to fill those roles (or how to find them).

The above operations and knowledge structures define a generic organiser role class. The *role* provides the ‘levers’ used to manipulate the composite. The responsibility of the organiser *player* that plays the organiser role is to decide what adaptive reconfiguration is necessary in response to changes in the environment, or to changes in the requirements imposed on the role the composite plays. Separating organiser-roles from organiser-players allows generic functions to be separated from the domain-specific decision-making process. Generic functions used to reconfigure the composite are generalised in an organiser role type. The decision-making functions that use domain-specific knowledge are implemented in the player. While, conceptually, the organiser player is separate from the organiser role, whether or not to implement them as separate runtime entities is a design decision. One alternative is to implement the organiser-player as a separate service. An advantage of decoupling the organiser role from its player is that players may vary in the intelligence they can apply to the decision-making process. For example, due to changing requirements an organiser player may be unable to reconfigure its composite in a way that fulfils its external obligations. In this case, the organiser can be upgraded to a more intelligent model such as a deliberative agent, human operator or programmer.

The organiser is responsible for monitoring the performance of the composite’s contracts either through notification by its contracts or by polling them. It also reconfigures the composite if environmental perturbation, or a change of requirements, leads to the composite not meeting (or potentially not meeting) its contractual performance obligations. The organiser can achieve reconfiguration by assigning and revoking contracts, and by changing the binding between functional roles and services as shown in Figure 8. For an organiser to be able to respond adaptively to changing situations, it needs to have at its disposal a range of services of various performance characteristics, and a mechanism for

service discovery such as enriched UDDI-like service registry. The discussion of such mechanisms is beyond the scope of this paper.

The management interface of a composite is the external interface of its organiser role. This is a conceptually similar arrangement to an ‘out-of-band’ manageability interface in MoWS (OASIS, 2005) which could be used for this purpose. Required and actual performance-measures pass backward and forward over this link. The organisers of the self-managed composites form a hierarchy of organisers – each organiser is responsible for the configuration of its composite and reports to the organiser up the hierarchy. This network of organiser roles (the thick lines in Figure 8) form a coordination system over which non-functional requirements (NFRs) and performance data flows (Colman and Han, 2005a).

If this organiser has difficulty organising the composite to meet the performance targets set by the enclosing composite, it escalates the warning up the chain of the coordination system hierarchy. Such escalation can be viewed as an organised form of exception-handling where performance messages and warnings flow through the coordination-system. These messages allow organisers, at the appropriate level, to reconfigure the organisational structure before error-messages are generated (as is the mechanism in BPEL).

3.5 Adaptive behavior of service composites

To illustrate the adaptive behaviour of a service composition, let us consider our example in Figure 8. The organiser wdo of the Widget Department, receives changes of requirement (NFRs) from the organiser of the enclosing self-managed composite (mdo in the Manufacturing division). This requirement involves the speedy delivery of a large quantity of widgets. This in turn requires a faster supply of Thingies (which are made into Widgets by Assembler a1) than specified by the contract between Assembler role (a1) and the role ThingyMaker (tm1) that is proxy for the external supplier service of Thingies (SystemX). The organiser wdo has a number of choices to meet the new requirement: it can change the binding between role tm1 and Web service and select alternative service SystemY; it can use two Thingy supplier services by creating another role instance tm2 and binding it to SystemY as in Figure 9; or it can attempt to

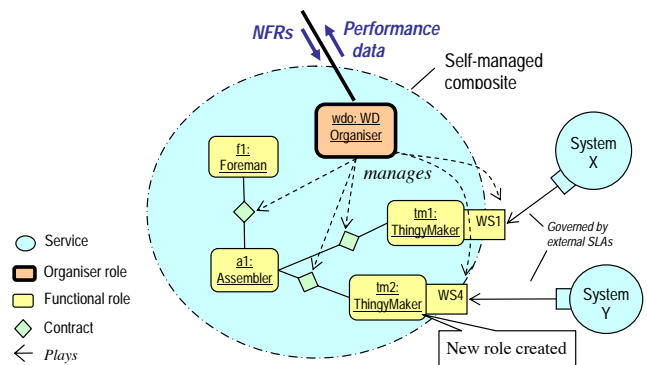


Figure 9 Adaptation through reconfiguration – managing indirection of association

renegotiate the external contract with SystemX to make it consistent with the internal contract requirement. A similar need to reconfigure the organisation of the Widget-department composite may arise if SystemX's performance decreases and it is no longer meeting its performance requirements as defined by the internal contract.

If the organiser wdo is unable to reorganise the composite to meet the new requirement or a change in environment, then the organiser (mdo in Figure 8) of the enclosing composite (the Manufacturing division) needs to reorganise its own structure. The organiser mdo could, for example, replace the Widget department composite with an outsourced service that supplies ready-made Widgets. In this way the application, which is made up of a nested hierarchy of self-managed composites, adapts to meet changes in requirements or changes in the performance of its external services.

A generalised example of a decision making process performed by an organiser player is illustrated in Figure 10. The organiser receives NFRs for the composite from the organiser of the enclosing composite (e.g. the Organiser mdo in Figure 8). It needs to translate these composite NFRs into NFRs of terms in the contracts it controls. The organiser also monitors the actual performance of its contracts either actively (polling) or passively (waiting for a notification from the contract). If there is a mismatch between the actual and required performance (either because the requirements in the contract have been changed, or because the actual performance has decreased), the organiser attempts to mitigate this underperformance by reorganising its composite.

The organiser chooses a strategy determined by the actual and claimed performance of the existing service and other available services. The actual-performance of a role-service pair is the historical performance measured as interactions pass through the contract. If a new service is allocated to the

role, then this actual performance data must be reset. In the absence of historical performance data, claimed-performance information could be obtained from a specification provided by the builder of the service or a third party accreditor.

In Figure 10, the types of function that may need to be implemented in an organiser player at the application-domain level (rather than the organiser role) have been italicised. These functions require some deliberative capacity. Depending on the type of application they might include:

- Translation of non-functional requirements (NFRs) that are provided by the enclosing composite, to NFRs for contracts within the organiser's composite. For example, a WidgetDeptOrganiser player needs to translate a requirement for widget throughput into NFRs of contract terms related to thingy production.
- Ability to discover services that are candidates to play roles.
- Ability to compare candidate service NFR characteristics (claimed and actual performance, availability etc.) and decide on the appropriate functional role-service bindings
- Ability to negotiate SLAs between the application and external services.
- Ability to decide on appropriate configurations of its composite (i.e. what roles and contracts to instantiate) in order to remediate changing NFRs or changing performance of its services. Any such reorganisation must maintain the composite's viability. For example, if the structure is based on a bureaucracy, the organiser must ensure proper chains-of-responsibility (i.e. supervisor-subordinate chains) are maintained to preserve the functional flow-of-control.
- If the system is a control system that is dynamically responding to perturbations in the environment, unstable

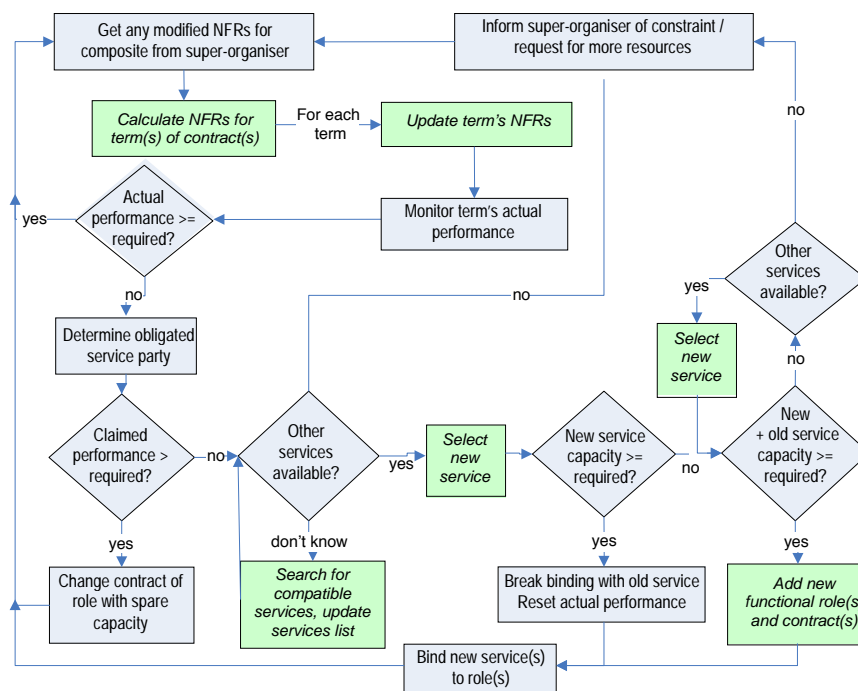


Figure 10 Example of a decision making process of an organiser player

feedback loops can be created. The organiser may need to regulate NFRs to dampen such oscillations.

The extent to which a player needs to implement the above functions will depend on the type of system being implemented. In general, the more uncertain the environment the more capability the organiser-player will need to maintain the viability of its composite.

4. IMPLEMENTATION

Our implementation of the ROAD framework allows management structures (roles, contracts, organisers and composites) to be implemented independently from the services that the application manages. All messages between services that are bound to the organisation pass through their respective roles and are intercepted by the contracts that associate those roles. As well as providing an abstract service definition (an aggregation of all the functional and non-functional requirements defined in the contracts the role is party to), the role acts as a message buffer (in case there is no service currently attached to the role) and router. Services only know about the role they play; they are “blind” to the structure between roles. Consequently, the role is responsible for routing out-going messages to an appropriately contracted associate. In the ROAD framework, a domain-specific functional role is created by extending an abstract role class that has the above characteristics.

ROAD contracts have been implemented using *association aspects* (Sakurai *et al.*, 2004) – an extension to the AspectJ language and compiler. This extension allows aspect instances (contracts) to be created that associate groups of objects (roles).² These aspects intercept all messages between roles and can prevent uncontracted roles talking to each other. This interception is used to ensure that all communication between roles accords with their contracts. The aspect pointcuts also allow points in an interaction to be defined where performance can be measured by calling an associated utility function. Contract instances can be created, or destroyed by the composite’s organiser. A detailed description of the implementation of ROAD contracts can be found in Colman and Han (2005c). The ROAD framework also provides abstract classes that can be extended by the application programmer to implement organisers and self-managed composites. A description of the framework library can be found in Colman and Han (2005a).

To test the framework, we have implemented a relatively

2. Aspects, as they are currently implemented in AspectJ (Eclipse Foundation, 2004), do not easily represent the behavioral associations between objects (Sullivan *et al.* 2002). To use aspects to create ROAD contracts, we want to create many instances of the contract type, each with its own state. Unlike classes, standard AspectJ aspects are not created using the new expression. By default, a singleton instance of the aspect is created when the aspect’s class file is loaded. This means exactly one instance of the aspect crosscuts the entire program. It follows that standard aspects are an unsuitable construct for creating multiple contracts of the same type. Alternatively, current implementations of AspectJ provide instantiation of perobject aspects. These can be used to associate a unique aspect instance to either the executing object (perthis) or the target object (pertarget). When an advice execution is triggered, the system looks up the aspect instance associated with that object and executes that instance. This allows the aspect to maintain a unique state for each object, but not the state of the associations between objects. In order to implement performance contracts, we need aspects instances that bind groups of objects, and that can be created and destroyed in the same way that objects are created and destroyed. Association aspects provide this capability.

closed application, i.e., the reference example used in this article (Linh *et al.*, 2005). Variable performance of services and changing non-functional requirements are simulated in order to demonstrate the adaptability of the system. In the test application, performance measurement has been limited to time-based utility functions. A simple mitigation strategy similar to that illustrated in Figure 10 has been implemented in an organiser player. This test organiser player is provided with a list of potential services.

A limitation of the current implementation is that, while new contract instances can be added to an application at runtime, new contract *types* cannot be created on the fly. This prevents runtime *functional* (as opposed to non-functional) recomp-osition *within* a composite. This limitation results from the compile-time weaving of the contracts into the functional code. However, as composites are themselves players, new composites with different internal functional configurations can always be swapped at runtime.

Sakurai (Sakurai *et al.*, 2004) has shown that there is little additional overhead in the use of *association-aspects* viz ordinary AspectJ aspects, thus there is little overhead in the basic interception mechanism of the contracts as they are currently implemented. This has been confirmed in our test implementation, with the overhead incurred by message interception in the contracts being relatively insignificant compared to that incurred in the Java method to SOAP message conversion. However, it remains to be seen whether a domain-specific coordination system based on association-aspects would impose a significant run-time penalty, particularly if complex utility functions are used and need to be calculated with each interception.

5. RELATED WORK

The relationship of the ROAD framework to WS* standards has been noted in various places above, and we will make only some general comments here. Like BPEL, our framework is a means of creating executable compositions, but ROAD has roles as its fundamental unit rather than activities. ROAD also allows the separation of management code from functional code which is otherwise tangled. The use of contracts in our framework allows a more natural mapping to inter-organisational business contracts than do process-based approaches. ROAD builds on the basic WS standards such as WDSL but also overlaps with a number of other extended standards that are not yet fully realised: in particular service selection (OASIS and UDDI Consortium, 2004), coordination (BEA Systems *et al.*, 2004; W3C, 2004), management (OASIS, 2005) and the definition of contracts (Global Grid Forum, 2004). As most of these standards address the external relationships between Web services rather than the internal composition, they complement rather than conflict with the ROAD framework.

While BPEL is not in itself adaptive, there has been much recent focus on making service composition more flexible. A recent overview of dynamic workflow-based composition can be found in Zirpins, Lamersdorf *et al.* (2004). These approaches focus on adaptive processes using process abstraction, rather than an adaptive role structure as presented here. Monitoring of services has also been addressed in Baresi, Ghezzi, *et al.* (2004) using external ‘Smart Monitors’

services rather than application-based monitors. However, there is no mechanism for ‘monitoring the monitors’ (organisers and contracts) as in the recursive ROAD structure. Ludwig, Dan, *et al.* (2004) propose the Cremona framework that addresses many of the same issues as the ROAD framework. This framework differs from ROAD in that it is based on WS-Agreement, and it focuses on external contracts. These contracts do not control interaction as the management level of a ROAD contract does.

While we have only briefly discussed the implementation of the ROAD framework in this paper, (Colman and Han, 2005c) gives a description of an aspect-oriented implementation of ROAD contracts. A similar aspect-oriented approach can be found in Baligand and Montfort (2004), where policies are woven into the Web service stubs rather than used to create instances of contracts.

6. CONCLUSION AND FURTHER WORK

The ROAD adaptive framework for Web services can be used by an application programmer to create applications that respond to changes in performance or other non-functional requirements. The application can also adapt to performance variations in the Web services it uses. In ROAD, the coordination and management code (organisers and contracts) is kept separate from the functional code (roles and services), and can be imposed on the functional code post-facto. The abstract management contracts can define interaction protocols and enforce these protocols. Concrete functional contracts inherit coordination capability from these abstract contracts, and are used to monitor performance. The framework provides general time-based performance measurement. The programmer can also extend the framework to define application specific metrics. The ability to extend contracts to cover application specific concerns is a major advantage of this approach over external horizontal standards. Organisers provide adaptive behaviour to the application by creating and revoking contracts between roles, and by binding roles to services. Organisers control a cluster of roles, contracts and service bindings called a self-managed composite. These composites are themselves services and may be distributed. Self-managed composites form a recursive hierarchy that distributes and localises management down through the application structure, while facilitating the fulfilment of system level requirements.

Adding adaptive indirection to applications complicates the programming task. The ROAD approach requires that service and role interfaces are compatible, and the signatures of these interfaces are captured in the contracts that bind the roles. These interdependencies mean that coding roles, contracts and service interfaces as separate classes duplicates interface definitions and can potentially lead to inconsistencies. An underlying meta-model of these interdependencies has been developed, and we are currently using this model as the basis for an Eclipse plug-in that will allow the developer to visually design organisational structures. Starting from the required and provided interface definitions of services, the programmer can select a subset of signatures from which to generate a role. Basic contract stubs are created by linking

roles and matching the required and provided signatures. Alternatively, roles can be defined from scratch, and then the player interface stubs are automatically generated. The aim of this on-going work is to reduce the complexity of programming adaptive applications, and to ensure consistency in the articulated structure.

The role-based nature of ROAD contracts allows a natural mapping to business entities. However, further work needs to be done on the mapping of the internal contracts defined in ROAD to the external associations between the application and the Web services it uses. In particular it would be useful to more formally define the relationship to WS-Coordination and WS-Agreement. A WSDL management interface could also be defined for composite organisers to make it compatible with WSDM-MoWS. This would allow the application’s organisational description to be distributed to service composites. Other management issues, such as a framework for organisational resource allocation, have yet to be addressed.

REFERENCES

- Alonso, G., Casati, F., Kuno, H. and Machiraju, V. (2004) *Web services concepts, architectures and applications*, Berlin, London: Springer.
- Baligand, F. and Montfort, V. (2004) A concrete solution for web services adaptability using policies and aspects. In *Proc. of the 2nd Inter. Conf. on Service Oriented Computing (ICSOC'04)* New York, NY, USA, ACM Press.
- Baresi, L., Ghezzi, C. and Guinea, S. (2004) Smart Monitors for Composed Services. In *Proc. of the 2nd Inter. Conf. on Service Oriented Computing (ICSOC'04)* New York, NY, USA, ACM Press.
- BEA Systems, IBM and Microsoft (2004) Web Services Coordination (WS-Coordination) <http://msdn.microsoft.com/library/enus/dnglobspec/html/WS-Coordination.pdf>.
- BEA Systems, IBM, Microsoft, SAP, A, and Siebel Systems (2003) *Business Process Execution Language for Web Services (BPEL4WS)* <http://ifr.sap.com/bpel4ws/index.html>.
- Beugnard, A., Jézéquel, J., Plouzeau, N. and Watkins, D. (1999) Making Components Contract Aware. *IEEE Computer* vol 32, no 7, pp 38–45.
- Bracciali, A., Brogi, A. and Canal, C. (2002) Dynamically Adapting the Behaviour of Software Components. In Arbab, F. and Talcott, C., (Eds.) *Proc. 5th Inter. Conf. on Coordination Models and Languages (Coordination'02)* LNCS 2315 York, UK, Springer.
- Colman, A. and Han, J. (2005a) Coordination Systems in Role-based Adaptive Software. In *Proc. of the 7th Inter. Conf. on Coordination Models and Languages (COORD 2005)*, LNCS 3454 Namur, Belgium, LNCS 3454.
- Colman, A. and Han, J. (2005b) Operational management contracts for adaptive software organisation. In *Proc. of the Australian Software Engineering Conf. (ASWEC 2005)* Brisbane, Australia, IEEE.
- Colman, A. and Han, J. (2005c) Using Associations Aspects to Implement Organisational Contracts. In *Proc. of the 1st Inter. Workshop on Coordination and Organisation (CoOrg 2005)* Namur, Belgium.
- Eclipse Foundation (2004) AspectJ, <http://eclipse.org/aspectj/>, Last accessed: Oct 2004
- Global Grid Forum (2004) *Web Services Agreement Specification (WS-Agreement)* www.gridforum.org/Meetings/GGF11/ Documents/draft-ggf-graap-agreement.pdf.
- Han, J. and Ker, K.K. (2003) Ensuring Compatible Interactions

- within Component-based Software Systems. In *10th Asia-Pacific Software Engineering Conf. (APSEC 2003)* Chiang Mai, Thailand., IEEE Computer Society.
- Linh, D.P., Colman, A. and Han, J.** (2005) The implementation of message synchronisation, queuing and allocation in the ROAD framework *Technical Report SUT.CeCSES-TR009*, Faculty of ICT, Swinburne University of Technology.
- Ludwig, H., Dan, A. and Kearney, R.** (2004) Cremona: an architecture and library for creation and monitoring of WS-agreements . In *Proc. of the 2nd Inter. Conf. on Service Oriented Computing (ICSOC'04)* New York, NY, USA,
- OASIS** (2005) Web Services Distributed Management-Management of Web Services 1.0, OASIS Standard, 9 March 2005 <http://docs.oasisopen.org/wsdm/2004/12/cd-wsdm-mows-1.0.pdf>.
- OASIS and UDDI Consortium** (2004) UDDI Specification Version 3.0 <http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm>.
- Ortiz, G., Hernández, J. and Clemente, P.J.** (2004) Web Service Orchestration and Interaction Patterns: an Aspect-Oriented Approach. In *Proceeding of 2nd Inter. Conf. on Service Oriented Computing (ICSOC)* New York, USA.
- Plasil, F. and Visnovsky, S.** (2002) Behavior Protocols for Software Components. *IEEE Transactions on Software Engineering*, IEEE Press vol 28, no 11, pp 1056-1076.
- Sakurai, K., Masuharat, H., Ubayashi, N., Matsuura, S. and Komiya, S.** (2004) Association Aspects. In *Proc. of the Aspect-Oriented Software Development '04* Lancaster U.K, ACM.
- Sullivan, K., Gu, L. and Cai, Y.** (2002) Non-modularity in aspect-oriented languages: integration as a crosscutting concern for AspectJ. In *Proc. of the 1st Inter. Conf. on Aspect-oriented software development, AOSD 02* Enschede, The Netherlands, ACM Press.
- W3C** (2004) Web Services Choreography Description Language (WS-CDL) Version 1.0 WD-ws-cdl-10-20041012 <http://www.w3.org/TR/ws-cdl-10/>.
- W3C** (2005) *Web Services Description Language (WSDL) Version 2.0 Primer* WD-wsdl20-primer-20050510 <http://www.w3.org/TR/wsdl20-primer/>.
- Zirpins, C., Lamersdorf, W. and Baier, T.** (2004) Flexible coordination of service interaction patterns. In *Proc. of the 2nd Inter. Conf. on Service Oriented Computing (ICSOC'04)* New York, NY, USA , ACM Press.