

Adaptive Application-Specific Middleware

Alan Colman, Linh Duy Pham, Jun Han, Jean-Guy Schneider
Faculty of Information and Communication Technologies
Swinburne University of Technology
Hawthorn, Australia
{acolman,lpham,jhan,jschneider}@ict.swin.edu.au

ABSTRACT

The open/dynamic environment of Service-Oriented Computing requires middleware that can cope with services that are heterogeneous, and possibly unknown, unreliable or untrusted. Service-oriented middleware also needs to support both, ad-hoc and long-lived relationships between such services, and provide mechanisms for service coordination and cooperation. This needs to be achieved in a rapidly changing technical context with standards that are continually changing and evolving. This paper introduces adaptive application-specific middleware composites which are built using the ROAD framework. These composites are adaptive runtime role structures that allow services to be composed and autonomously reconfigured. In these composites, dynamic contracts control interactions between services, set non-functional requirements for those interactions, and measure the QoS of services against those requirements. These middleware composites can themselves be encapsulated as services that can be recursively composed and distributed. These composites can cope with changing requirements and performance of the services they compose. Composite roles and contracts also map naturally to business entities.

Categories and Subject Descriptors

D.2.11 [Software Engineering]: Software Architectures—*Domain Specific Architectures*; D.2.12 [Software Engineering]: Interoperability—*Distributed Objects*.

Keywords

Adaptive Middleware, Service-Oriented Computing, ROAD

1. INTRODUCTION

Middleware standards and technologies are, in general, generic. By using a common middleware, heterogeneous applications are able to communicate and collaborate. Middleware technologies also hide complexity and add value to

these interactions. Taking a high-level, Enterprise Application Integration (EAI) view, Figure 1 illustrates a simple schema of two heterogeneous applications that communicate via a conventional middleware layer that handles various properties related to their interaction (reliable messaging, logging, persistence etc.).

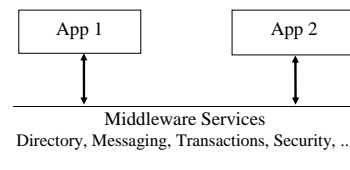


Figure 1: Conventional Middleware.

In more open environments typical of service-oriented computing, a number of shortcomings of the conventional view of middleware become apparent, particularly when the middleware concept is applied in cross-organisational Web services rather than conventional middleware integration within a single organisation. Cross-organisational composition of services involves issues such as lack of trust and asynchronous, long-lived transactions, as well as deployment issues such as the location of the middleware in a distributed system [1].

The other major shortcoming of the conventional middleware concept is the necessity for all participants of an interaction to agree on, and use, interoperable standards. In the fast-changing world of Web services, the basic technology for handling interactions is well established and accepted, but standards for handling more complex interactions such as WS-Coordination, WS-Agreement, and OWL-S are still evolving and are sometimes overlapping depending on the domain and the originating standards body (e.g., W3C, OASIS, Global Grid Forum, etc.). In order to build adaptable applications in this changing and uncertain technical context, it would be desirable for applications to be able to make use of heterogeneous standards, while not being bound to any one standard. Although we cannot do without standards, the challenge remains to create integrated applications that can make use of the heterogeneous middleware technologies, but that are not dependent on any particular technology. To use an analogy: programming languages like Java can run on heterogeneous operating systems by providing an independent layer (Bytecode running on the JVM) between the application code and the operating system. In order to preserve this *Write Once, Run Anywhere* approach to software development, we need to avoid tightly coupling

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MW4SOC '06, November 27-December 1, 2006 Melbourne, Australia
Copyright 2006 ACM 1-59593-425-1/06/11 ...\$5.00.

applications to their execution environment, whether that environment be an operating system or middleware.

In this paper, we introduce the concept of *application-specific middleware*. As in conventional middleware, all messages between the component services pass through a middleware layer. In addition, application-specific middleware provides adaptive structures for the composition, control of service interactions, and the measurement of the Quality of Service (QoS) of those interactions. Application-specific middleware performs no domain-specific function by itself, instead it provides abstract functional roles that can be played by other entities. These roles are formed into adaptive structures, tailored to the particular application, for the composition and control of service interactions.

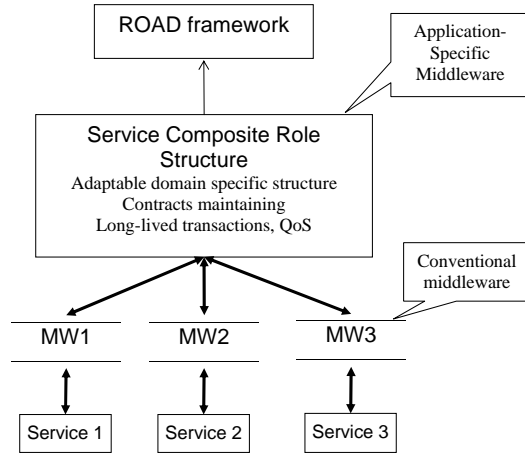


Figure 2: Schema service composite as application-specific middleware.

As shown in Figure 2 above, application-specific middleware can be viewed as an extra layer that provides a level of indirection and management between services. This middleware consists of a composite(s) of dynamically contracted roles which are played by the various services in the composition. This composite role-structure is autonomously managed by a composite *organiser*. The roles, contracts and organisers that constitute the application-specific middleware are built from a generic library called the ROAD (Role-Oriented Adaptive Design) framework [3, 5]. The services that play roles can use whatever middleware standards they are built to, provided the service composite has adaptors that support those standards. Therefore, the composite functions as an interoperability bridge [7].

Application-specific middleware in the form of role-structures provides a number of advantages for service-oriented computing in open and dynamic environments, including the ability to:

- dynamically compose heterogeneous services by creating role instances, associating those roles with contracts, and binding concrete services to those roles,
- create contract instances that specify permitted interaction patterns (e.g., business protocols) and control actual interaction between specific instances of roles (as distinct from policies that apply to all instances of a type),

- specify non-functional requirements (e.g., QoS) on contracted interactions between roles, and to measure the performance of interactions with respect to those requirements,
- dynamically alter those requirements and add new requirements,
- autonomously reconfigure the relationships between services in response to changing requirements or changing environments (e.g., dynamic selection of services to optimise some QoS criteria), and
- create both ad-hoc and long-lived relationships between services, and to store the state of those relationships; that is, to implement a *virtual organisation*.

The rest of this paper is organized as follows: in Section 2, we will introduce, as a motivating example, an application-specific middleware that assists institutional clients (i.e. libraries) to purchase books from Web-Service suppliers. Section 3 briefly discusses the function and structure of composite role structures as defined in the ROAD framework, and how these composites can be recursively composed and distributed. We then show how a middleware composite adapts itself in response to changing requirements and the changing performance of the services attached to the composite. In particular, we examine how QoS and other non-functional properties can be handled. Section 4 discusses some issues that arise from this approach and Section 5 discusses related work. Section 6 concludes with a summary of the main observations and future work.

2. A MOTIVATING EXAMPLE

In this section, we will illustrate the concept of application-specific middleware by describing a middleware role structure that mediates between purchasers and suppliers of books. This structure allows services of various types (clients, suppliers, brokers) to be dynamically attached and regulated so that clients can purchase books from suppliers on the best terms.

Large institutional libraries purchase many books from many suppliers. The same book can often be sourced from multiple suppliers. Because libraries are such large purchasers of books, suppliers have an interest in providing a high level of service to such clients (competitive prices, quick turn-around etc). Likewise, libraries have an interest in maintaining good working relationships with key suppliers. As distinct from an individual purchase transaction, the terms of a library-supplier relationship are more long-lived, but the properties of the relationship often vary. The corresponding variables (we will call them *terms-of-trade*) could include discount, if any; time from order to delivery; terms of payment (prepay, on invoice, days to pay); past reliability of supplier; value of trade with supplier; reputation of supplier (particularly if purchases are prepaid); and duration of relationship. A library will have desired values for this set of variables: low price, payment 30 days on statement, delivery not more than 2 weeks, etc.

Application-specific middleware will provide a structure that allows libraries, book suppliers and broking services to play role(s) in that structure. A library service submits requests for quotes and book orders over its service interface. Book suppliers provide automated Web service interfaces for

book search, quoting, ordering, and payment. The protocols for these transactions and the terms of trade vary between each supplier, and the broker needs to be able to work with all of them. The responsibility of the broker is to purchase the books needed by the library, with optimal terms of trade for the library, while maintaining strong relationships with suppliers. Therefore, the broker has to assess the value of each transaction to the library (not just the price). It also has to ensure that the library and supplier have matching order, payment and delivery protocols. In practice, there are many different protocols that client, broker and supplier can follow, for example, payment before delivery, payment after delivery, payment on invoice, and client either pays supplier via broker or directly, etc. The application-specific middleware has to be flexible enough in order to accommodate these variations and enforce them at runtime.

In assessing the library’s optimal terms of trade, the broker has to abide by the purchasing policies and preferences set by the library. These preferences often have to be traded-off. For example, the broker can source books either locally or from overseas. While overseas books are often cheaper, delivery times are typically longer. Methods of payment also vary between local and overseas suppliers. Most (not all) local suppliers are happy to supply books on invoice as the library is a reliable payer. Overseas suppliers invariably want payment up front. The library therefore sets a purchasing policy (*general terms-of-trade*) that provides rules to the broker so that it can trade-off these variables. For example, it instructs the broker that it prefers to pay on invoice, but that it is prepared to pay upfront if the total cost saving is greater than 15%, provided the supplier has a good reputation. As well as these general terms-of-trade, a library will have specific relationships that have been negotiated with each of its suppliers. The *specific terms-of-trade* will define the rules for transactions between those parties (protocol for payment, discounts, etc.). The advantage for a library using a broker is that the broker hides many of the details of relationships with suppliers, and automates the supplier selection process. A library, broker and the suppliers can be thought of as forming a *virtual enterprise* that creates and maintains dynamic business relationships of varying duration between its entities.

3. MANAGED ROLE STRUCTURES

In this section, we will show how the above scenario can be represented by managed role composites built using the ROAD framework, and discuss the characteristics of those composites. Figure 3 shows a role structure that represents the organisational relationships within a **Broking Composite** between clients, brokers and vendors. Client role instances are created for each library. Each book seller service is represented by a different role, rather than having a single vendor role attached to different seller services at different times.

Multiple vendor and broker roles are required because each broker instance represents a particular library client, and the relationship between a broker and vendor contains the specific terms-of-trade and the history of actual ‘performance’ of each parties with respect to the relationship. This is because the agreed *specific terms-of-trade* and history of transactions needs to be persistent for each broker-vendor relationship. Note that it is possible for a concrete service (e.g. the Broking Agent Service) to play multiple roles (i.e. **Broker1** and **Broker2**) in an application-specific middleware

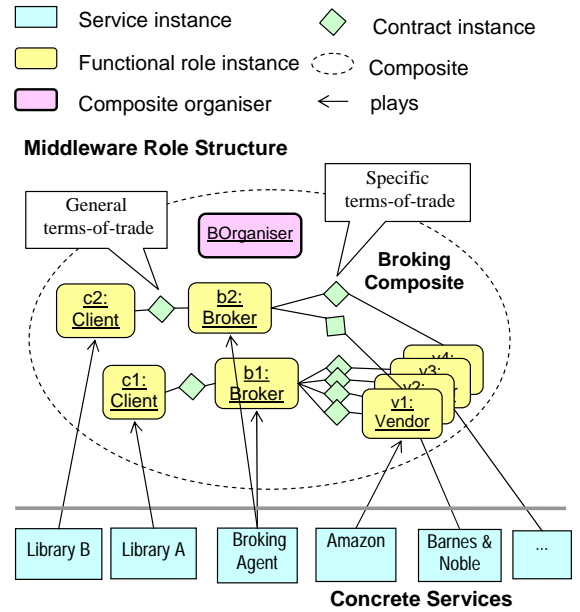


Figure 3: Broking Composite with multiple clients.

composite.

As discussed in Section 1, an application-specific middleware composite does not perform any domain function by itself, rather it defines and maintains *specific* relationships between concrete services via their roles. In our example, the client, broker and vendor roles are placeholders that describe a ‘position description’ within an organisational structure. Likewise, the terms of contracts can be defined declaratively. Both contract and role descriptions can be passed to a middleware composite factory that dynamically creates the runtime organisational structure. In the following, we will discuss the properties of roles, contracts, organisers and composites that make up this middleware layer.

Application-specific middleware can be created using the ROAD (Role-Oriented Adaptive Design) framework [3, 5]. The ROAD framework is a set of abstract Java classes used to define adaptable software “organisations.” These abstract classes are extended by application developers to create a software system that can autonomously adapt both to changes in the non-functional requirements, and to changes in the performance of the functional services or other functional software entities that are bound to the organisational structure.

In ROAD, *functional* roles are first-class runtime entities that hold abstract service definitions. Because the services that play instances of roles can be transitory (for example, there may be no service currently available to play a role), roles store incoming message in queues. Roles also perform the function of message routers as the services that are bound to a role in an organisational structure do not directly reference each other. Roles may be contracted to a number of other roles (e.g., a broker role instance in Figure 3) and, therefore, need to forward messages from their service-player to the appropriate associated role. While roles are implemented in Java, they may need to communicate with a heterogeneous range of services using different conventional middleware technologies, and adaptors are bound

to roles as needed. From the role’s viewpoint these adaptors are proxy services that hide the heterogeneity between different services. They convert messages from the middleware’s native Java to the appropriate format (e.g. SOAP if the remote service is a Web service) and provide other properties needed for interoperability (e.g. WSDL service endpoint references) [4].

Contracts perform three functions in a ROAD organisational role structure: composition, interaction control, and performance monitoring. A ROAD contract consists of one or more *terms* that define a dynamic, rich association between two roles. Contracts are implemented using *association aspects* [5, 14], an extension to the AspectJ compiler. Such association aspects allow contract *instances* to be created that associate groups of objects (in this case associations of role instances). By creating and/or revoking contract instances, the topology of the composition of roles can be altered. As all roles (as opposed to services) are internal to the organisation, ROAD contracts are also internal to the organisation. All runtime communication between functional services bound to the organisation is via such contracted roles. ROAD contracts (being association aspects) have the ability to intercept the communications between these roles using *pointcut* pattern matching on method signatures. This interception is used to prevent communication not authorised by the contract, and to invoke aspect *advice* before and after a transaction, to measure the performance of the transaction relative to a term in the contract. In this way contracts perform a similar function to *interceptors* in conventional middleware. Contract terms define the mutual obligations of the participant roles in an organisational context. They define the interactions that are permissible or required by the participant roles, and can be used to enforce sequences of interactions. Contracts can also be used to set arbitrary non-functional requirements in the form of utility objects on their roles’ interactions (like *SLA parameters* in [10], these are separable from the contract terms). Contracts monitor those interactions for compliance to their respective requirements.

Organisers create and destroy roles, make and break the bindings between organisational roles and services (service selection), and create and revoke the contracts between the roles. They can thereby create various configurations of roles and services. Organisers set the performance requirements for the contracts they control, and receive performance information from those contracts. Organisers have reconfiguration strategies they can employ if they detect under-performance in the composite they control. In short, organisers provide the adaptivity to the composite application by managing the composition and instantiation. Organisers, along with the contracts and roles that they control, can be viewed as a middleware management layer that composes and controls the interaction of the functional services.

Each organiser is responsible for the configuration of a set of roles and contracts: a *self-managed composite*. In terms of a management analogy, a self-managed composite in a business organisation would be a department. An *instantiated middleware composite* can itself be a service as the roles internal to the composite have services attached. Messages to the composite are delegated to its internal role-players. Middleware in the form of these self-managed composites can therefore be distributed (as any service can be distributed). As a composite can itself be a service that plays a role in

another composite, it follows that composites can also be recursively composed and/or decomposed.

The ability to compose/decompose middleware role composites enables complex systems to be modeled. For example, a library could itself be made up of a number of services that are organised by a role composite. In Figure 4, the broking composite instance is associated with two types of library services. The first type of library service (**LibraryA**) is itself a role composite (not all its internal roles or associated services are shown in the diagram). **LibraryA** plays the **Client1** role in the broking composite. Conversely, the broking composite plays the **Supplier** role in the **LibraryA** composite. The implication of this bi-directional role-playing is that all functional messages between the composites pass through these respective roles. Such a structure would suit composites in different organisational domains (e.g., with different owners), because each composite has an internal role that is a proxy for the other composite. Therefore, functional coupling is kept to a single interface.

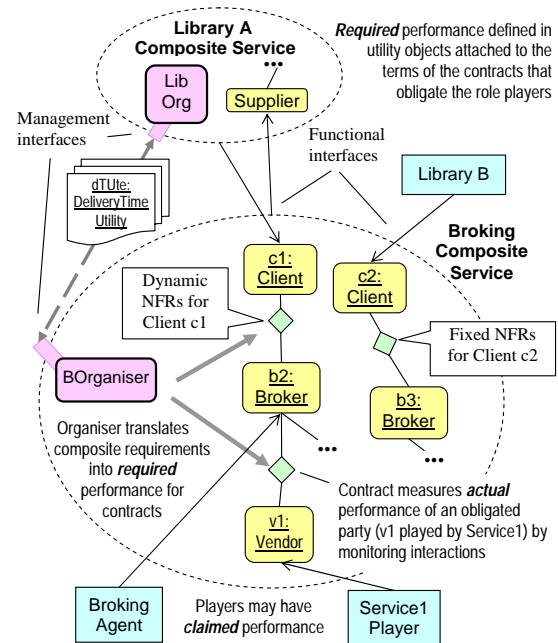


Figure 4: Composition of role composites, and the setting and measurement of NFRs.

Each self-management composite has exactly one organiser role (e.g. **BOrganiser** in the **Broking Composite**). A self-managed composite has a management interface which is the external interface of its organiser role, as shown in Figure 4. This interface performs a similar function to an “out-of-band” manageability interface in MoWS [13] in that required and actual performance-measures pass backwards and forwards over it.

The second type of library (**LibraryB**) in Figure 4 is not a role composite, but a basic service. It has a functionally compatible service interface through which it interacts with the broking composite, but it has no management interface or organiser. While both **LibraryA** and **LibraryB** can play roles in the broking composite (i.e. order books from the broker), basic services like **LibraryB** cannot dynamically communicate non-functional requirements (NFRs) to com-

posites such as the Broker. LibraryB's NFRs would need to be set statically in advance (or through some form of supervisory control via BOrganiser).

Application-specific middleware composites can adapt to changes in two ways: *regulation* and *reconfiguration*. Regulation involves the setting of non-functional requirements in the contracts it controls. Reconfiguration involves changing the role structure by creating/destroying roles and contracts, or swapping services that are bound to its roles.

A composite receives NFRs via its management interface. The organiser of the composite needs to convert these NFRs into contract terms (a process similar to *intra-service composition* in [15]). In Figure 4, as both the LibraryA and the Broking composite are self-managed composites, they both present a management interface over which changing NFRs flow between their respective organisers. In this case, the organiser within the libraries' acquisition departments sends the Broking Service NFRs relating to their general terms-of-trade. The organiser of the broking composite service (BOrganiser) stores these NFRs in the respective Client-Broker contracts, and then interprets them into specific terms-of-trade stored in the Broker-Vendor contracts. As shown in Figure 4, these NFRs are in the form of utility function objects. The expected value of the parameters can be set in these objects (e.g., for a DeliveryTimeUtility these values might be given for TargetMean, BreachThreshold, etc.). As well as storing the *required* performance, a utility object stores the actual performance based on the history of interactions between the contracted roles. Each utility class implements a method calculateUtility that returns a value associated with contract performance (e.g., InBreach, UnderPerforming, Performing).

Reconfiguration, the second type of adaptation, is needed when new services (Libraries or Book Sellers) are added to, or removed from, the composite. This can involve merely swapping the service an existing role is playing with another service, or adding/removing roles and creating the corresponding contracts. Reconfiguration can occur in response to new services being discovered, or in response to an existing service not meeting its contractual obligations as defined by the terms of its role's contract(s).

These changes to NFRs can only be triggered via the management interface, that is, either from another composite or by a supervisory control program that can access the management interface. The ability to trigger a change in requirements or structure within a middleware composite raises the issue of which services have the authority to make changes in the composite. In our example, we would not want a book seller arbitrarily changing the terms-of-trade with a broker or client, it follows that some scheme of ownership rights and security need to be established in middleware composites that operate in open environments.

In the case where adaptivity is required across an organisational boundary, one architectural solution would be for each organisation to deploy and maintain its own composite middleware. This would be the case if the LibraryA and the Broking Service had different owners (c.f. Figure 4). These middleware composites communicate via their management interfaces and the functional roles they play in each other (the Library composite playing the Client role in the Broking Composite, and the Broking Composite playing the Supplier role in the Library composite). As discussed before, there is no restriction on the distribution of composites. As these

roles can only communicate according to the rules in their contracts, each composite can prevent the other party invoking unauthorised interactions. The transmission of NFRs between composites via their management interfaces would need to be negotiated. The issue of contract negotiation is however beyond the scope of this paper.

4. DISCUSSION

By its very nature, application-specific middleware requires customisation. The ROAD framework assists this process by providing a library of abstract classes (roles, contracts and composites) that can be extended to suit the needs of the domain/application. These framework classes provide composition, monitoring, and adaptive mechanisms. An application developer can extend the classes with domain-specific roles, contracts and composites which can largely be defined declaratively. The library also provides a set of abstract contracts that define prototypical interaction patterns between roles (e.g. Buyer-Seller, Supervisor-Subordinate, Peer-Peer). The lack of space does not permit a description of the ROAD framework in further detail; interested readers are referred to [3, 4]. A 'proof of concept' framework and a number of test applications have been developed. Both the variable performance of services and the changing of non-functional requirements have been simulated in order to demonstrate the adaptability of the system. A simple mitigation strategy has been implemented in an organiser player which is provided with a predefined list of potential services (service discovery has not yet been implemented). Currently, our prototype only provides Web service (WSDL-SOAP) adaptors.

Controlling and monitoring interactions using an application-specific middleware composite has advantages and disadvantages. As described in Section 1, the composite can act as an interoperability bridge to which middleware adaptors can be added on an ad-hoc basis. However, if the two remote services that communicate via the middleware share the same standards (e.g., WSDL-SOAP) or are interoperable, then a relative performance overhead is incurred in converting messages to the composites' language (Java) and then marshaling them back to the original format. As this overhead can be quite substantial, the application of our approach is probably limited to domains that do not involve high loads or require rapid response times. Commercial transactions in virtual enterprises where long-lived transactions predominate would be an appropriate field of application. On the other hand, based on testing of the prototype framework, the overhead imposed by the ROAD framework in message interception, contract enforcement and updating the performance measurement is relatively insignificant compared to format conversion, accounting for less than 2% of the total overhead in a Web service context.

Application-specific middleware only deals with "internal" contracts between roles within the organisational boundary, although those roles are proxies that represent external services. This approach does not preclude the use of external contracts [9, 10, 15] or external coordination mechanisms [2]. The internal contract that binds the proxy role to the rest of the system can serve as an *agreement template* for defining external service-level agreements according to the WS-Agreement standard (or a *coordination context* in the case of WS-Coordination). The concept of a contract also maps naturally to Service Level Agreements at the business

level, in part because roles map better to business entities than, for example, BPEL activities.

5. RELATED WORK

Adaptive middleware has been an active area of research in recent years. However, much of this work focuses on the management of variable computational and network resources [6, 11] rather than focusing on adapting to changes in service-level requirements as described in this paper. Adaptive architectural frameworks such as Rainbow [8] have much in common with the approach described here in that management is exogenous to the services. However, Rainbow infrastructures have global adaptation strategies (rather than being modularised in a composite's organiser) and they cannot be recursively decomposed or distributed as can ROAD composites.

Application-specific middleware is capable of expressing service compositions. Like BPEL, our framework is a means of creating executable compositions, but with roles as its fundamental unit rather than activities. Another advantage of a role-oriented approach is that it allows for the separation of management code from functional code that might otherwise be tangled. The use of contracts in our framework also allows a more natural mapping to inter-organisational business contracts than do process-based approaches.

As mentioned above, ROAD complements work done on SLAs in Web services, [10, 15, 16]. In terms of frameworks, Ludwig et al. [12] have proposed the Cremona framework that addresses many of the same issues as the ROAD framework. Cremona differs from ROAD in that it is based on WS-Agreement, and it focuses on external contracts rather than contracts between roles within the middleware.

6. CONCLUSIONS

This paper introduces adaptive application-specific middleware composites which are built using the ROAD framework. These composites are adaptive runtime role structures that allow services to be composed and autonomously re-configured, support both ad-hoc and long-lived service relationships, and provide mechanisms for service coordination and cooperation. These composites can potentially act as interoperability bridges between services running on heterogeneous conventional middleware platforms. Dynamic contracts in these composites control interactions between services, set non-functional requirements for those interactions, and measure the QoS of services against those requirements at runtime. These composites can reconfigure themselves in response to changing requirements and measured performance of the services they compose. Such middleware composites can themselves be encapsulated as services that can be recursively composed and distributed. Composite services communicate changing NFRs and performance data over their management interfaces. Composite roles can be used to model business entities, and contracts also map naturally to cross-organisational service-level agreements.

Further work needs to be done in integrating application-specific middleware composites with standards for service discovery, coordination and the negotiation of service-level agreements. Research challenges also remain in how to dynamically generate adaptors to overcome the heterogeneity of service middleware technologies; how to represent protocols of services (required order of exchanged messages); how

to address different non-functional requirements such as security; and how to incorporate mechanisms for negotiation between composites belonging to different organisations.

7. REFERENCES

- [1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web services concepts, architectures and applications*. Data-centric systems and applications. Springer, Berlin, London, 2004.
- [2] BEA Systems, IBM, and Microsoft. Web services coordination (ws-coordination), 2004.
- [3] A. Colman and J. Han. Coordination Systems in Role-based Adaptive Software. In *Proc. of the 7th Int. Conf. on Coordination Models and Languages*, LNCS 3454, pages 63–78. Springer, 2005.
- [4] A. Colman and J. Han. Adaptive service-oriented systems: an organisational approach. *Int. Journal of Comp. Systems Science and Engineering*, 21(4):235–246, 2006.
- [5] A. Colman and J. Han. Using Associations Aspects to Implement Organisational Contracts. *Electronic Notes in Theoretical Comp. Science*, 150(3):37–53, May 2006.
- [6] H. A. Duran-Limon, G. S. Blair, and G. Coulson. Adaptive Resource Management in Middleware: A Survey. *IEEE Distributed Systems Online*, 5(7), 2005.
- [7] W. Emmerich. *Engineering Distributed Objects*. John Wiley & Sons, 2000.
- [8] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *IEEE Comp.*, 37(10):46–54, Oct. 2004.
- [9] Global Grid Forum. Web services agreement specification (ws-agreement), version 1.1, May 2004.
- [10] IBM. Web services level agreement (ws-la) language specification. Technical Report v1.0, ws-la-2003/01/28, IBM, 2004.
- [11] M. Leclercq, V. Quéma, and J.-B. Stefani. DREAM: A Component Framework for Constructing Resource-Aware, Configurable Middleware. *IEEE Distributed Systems Online*, 6(9), Sept. 2005.
- [12] H. Ludwig, A. Dan, and R. Kearney. Cremona: an architecture and library for creation and monitoring of WS-agreements. In *Proc. of the 2nd Int. Conf. on Service Oriented Computing (ICSOC '04)*, pages 65–74, New York, NY, Nov. 2004.
- [13] OASIS. Web Services Distributed Management – Management of Web Services, Version 1.0, Mar. 2005.
- [14] K. Sakurai, H. Masuhara, N. Ubayashi, S. Matuura, and S. Komiya. Design and Implementation of an Aspect Instantiation Mechanism. *Trans. on Aspect-Oriented Software Dev.*, 3880:259–292, 2006.
- [15] J. Skene, D. D. Lamanna, and W. Emmerich. Precise Service Level Agreements. In *26th Int. Conf. on Software Engineering (ICSE '04)*, pages 179–188, Edinburgh, UK, May 2004.
- [16] V. Tosic and B. Pagurek. On comprehensive contractual descriptions of web services. In *Proc. of IEEE Int. Conf. on e-Technology, e-Commerce and e-Service. IEEE '05.*, pages 444 – 449. IEEE, 2005.