Leveraging Modes and UML2 for Service Brokering Specifications

Howard Foster, Sebastian Uchitel, Jeff Kramer, and Jeff Magee

Department of Computing, Imperial College London,
180 Queen's Gate, London SW7 2BZ, UK
{hf1,su2,jk,jnm@doc.ic.ac.uk}

Abstract. A Service-Oriented Computing (SoC) architecture consists of a number of collaborating services to achieve one or more goals. Traditionally, the focus of developing services (as components) has been on the static binding of these services within a single context and constrained in an individual manner. As service architectures are designed to more dynamic, where service binding and context changes with environmental disturbance, the task of designing and analysing such architectures becomes more complex. UML2 introduces an extended notation to define component binding interfaces, enhanced activity diagrams and sequence diagrams. We propose the use of Modes to abstract a selected set of services, their coordination and reconfiguration, and use models constructed in UML2 to describe brokering requirements which can be synthesised for input to service brokers. The approach is implemented through the use of a Modes Browser, which allows service engineers to view specifications to a prototype dynamic service brokering agent.

1 Introduction

The area of service-oriented computing is exploiting the increase in use of distributed component architectural patterns, descriptions and service offerings. Describing such dynamic component configurations is complex, and traditionally system design has presented a static view of designing such systems with the expectation that the architecture does not evolve dynamically or can change due to environmental disturbances. As services are the means by which components interact, describing such changes is a highly important step in defining a service engineering perspective to systems development and requires an enhanced modelling notation to support such requirements. One such notation, Darwin, is a declarative binding language which can be used to define hierarchic compositions of interconnected components. The central abstractions managed by Darwin are components and service. In addition to its use in specifying the architecture of a distributed system, Darwin has an operational semantics for the elaboration of specifications such that they may be used at runtime to direct the construction of the desired system. More recently, UML2 has also introduced a number of closely related concepts to assist in describing the collaborative and distributed nature of components, and in particular has added notations

for component architecture constraints, enhanced activity diagrams to model workflow and enhanced sequence diagrams to address message invocation styles and alternative behaviours. UML has become a widespread, common practiced language to support software development and as such, is highly accessible by system engineers.

In this paper we present an abstraction of dynamic systems using the concepts of software modes, and we use UML2 elements to represent both the identification of Modes from a services domain (leading to a UML2 Modes Profile), the identification of self-management artifacts for operating these modes in a service-oriented architecture, and extracting specifications for dynamic service brokering requirements and capabilities. In section 2, we provide a background to SoC and discuss self-management techniques for distributed component architectures. In section 3 we illustrate a model-driven approach to engineering modes for SoC. In section 4 we provide an overview of the concept of Modes and also detail how modes are described in elements of the UML2 notation. Section 5 combines the concepts in a UML2 Extension Profile for modes, whilst in section 6 we detail how a Modes Model can be used to extract deployment artifacts for Service Brokers in SoC. Finally, in sections 7 and 8 we discuss limitations and provide an indication of future work direction.

2 Background

A mode, in the context of Service-Oriented Computing (SoC) abstracts a set of services that collaborate towards a common goal [5]. A mode can be used to identify which services are required in a service composition, and assist in specifying orchestration and choreography requirements through service component state changes. As a practical example, modes can assist by describing the requirements and capabilities for dynamic service brokers in an Service-Oriented Architecture (SOA) by describing both required and provided service characteristics in a particular system state. Modes also define an operating environment by way of architectural constraints and of component behaviour. They can specifically be used towards addressing reconfiguration issues within a self-managed system. Self-management typically is described as a combination of self-assembly, self-healing and self-optimisation. Self-management of systems is not a new idea, with ideas from both the cybernetics [16] and system theory [15] worlds. As discussed in [14] however, one of the main problems in self-management is to understand the relationship between the system and its subsystems: can we predict a system's behavior and can we design a system with a desired behavior? The need for such systems management is however more desired than ever before as we rely on distributed networks of interrelated systems through the extensive growth and increase in reliability of the internet. In SoC for example, a dynamic service brokering solution needs to address issues like how to specify the QoS requirements and capability, and how to select the most appropriate service provider among several functionally-equivalent providers based on the QoS offered. An example service broker engine is called Dino [11]. Dino provides

a runtime infrastructure consisting of a number of brokers. These brokers are responsible for, among other things, service discovery and selection on behalf of service requesters. Dino service requirements include both functional and QoS requirements. Similarly, every service provider needs to define its service capability (including both functional and QoS offerings) in the specification language provided by Dino. A service requester forwards its requirements specification document to a Dino broker at runtime, and thereby delegates the task of service discovery and selection to the broker. A shared understanding of the semantics of functional and QoS properties specified by service requesters and providers is achieved by referring to common ontologies.

Another key part in the self-management of a dynamic SOA is the runtime policy which specifies how and when reconfigurations occur. In [17], a policy set for SOA is described as being part of one of four categories for either process (the behaviour required), architecture (the configuration and reconfiguration of service components), operational (non-functional properties such as QoS levels) and relationship (trust and functional support). A policy for a system designed for an SOA architecture should consider each of these. Although policies clearly cover much more than QoS, in this paper we focus on the architecture constraints, and specifying QoS constraints on service operation and contracts for dynamic service brokering.

2.1 Related Work

Related work falls in to several categories. Firstly self-management of SOA's has attracted both policy and technology management work. In [8] for example, the authors describe a state-space modelling approach for system policies to facilitate self-management of enterprise services. Although concentrating on an enterprise model there is no running example of how this translates to a physical runtime usage. Whilst in [1] they focus on low-level administration of service networks to monitor service quality and react to notification events. Secondly, for modelling requirements of services and SOA there is also IBM's UML2 Profile for Software Services [6]. This profile provides a set of stereotypes that represent features for the service engineering life cycle, including a service specification (interface), gateway (ports) and collaboration (behaviour specifications). In [2] the authors present a profile aimed at specifying both architectural and behaviour aspects of SOA in UML for both web and non-web deployment environments. A particular difference in this work is that there is a focus on service messages and routing, whilst also being able to specify registry and adapter elements. The SENSORIA project has also extended UML for SoC specifications in [7], adding stereotypes for functional and non-functional requirements, contracts and behavioural modelling of services. Transformations of UML for SOA deployment artifacts has been discussed in [9]. Here the authors present an approach to transform UML use case diagrams to object diagrams for implementation (in further transformation approaches). Generally what is missing from these existing profile approaches is the ability to identify the requirements and capabilities of services and then to elaborate on the dynamic changes anticipated for self-management.

Our work expands on the ideas presented in these profiles by considering the more dynamic change of architectures, for self-management and brokering in a dynamic SOA.

3 Approach

To assist in defining modes of a service-oriented architecture, we have formed an approach known simply as "SelfSoC". SelfSoC comprises descriptions of architecture, behaviour and management, and we believe can be used to enhance the engineering of service brokering, orchestrations and choreography requirements whilst also describing the reconfiguration policies of service systems for runtime.

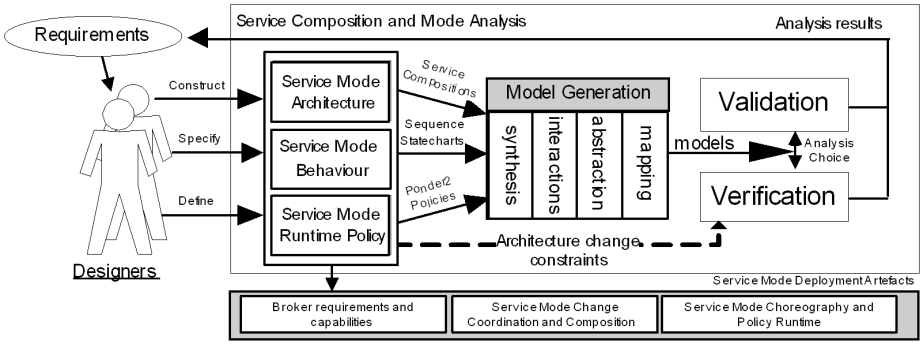

Fig. 1. Service Composition, Analysis and Deployment Artefacts in SelfSoC

Figure 1 illustrates the overall approach, and the areas considered in SelfSoC. In this paper we specifically focus on the areas of mode component architectures, service constraints and extracting broker requirements. Each area is considered under a number of mode elements. Firstly, the service modes architecture is specified as components and composite components defining the service capabilities and relationships between them. Constraints are specified around these relationships, as part of a runtime policy, as to when and how these bindings can occur. The behaviour area of our approach considers the scenarios of service usage (a mode can be referred to in one or many scenarios), and the interaction sequences permissible in certain modes. Thirdly, the area of service mode runtime policy considers the runtime management of services and reconfiguration between different modes. The aim of the approach is to accept descriptions of these three core specifications and generate appropriate models for analysis and generation of deployment artifacts. As a first step towards providing this approach, we concentrate on the service mode architecture description and generating broker requirements and capabilities without the specifications of mode behaviour and a full runtime policy. Analysis of mode architectures and service behaviour policy is left as a future work which is discussed in section 8.

3.1 Case Study

Our case study is based upon a set of requirements formed from those reported as part of a European Union project called SENSORIA, our role is to support the deployment and re-engineering aspects of this case study and in particular, to provide a self-management approach. In this case study are a number of scenarios relating to a Vehicle Services Platform and the interactions, events and constraints that are posed on this services architecture (illustrated in Figure 2). One particular scenario focuses upon Driving Assistance, and a navigation system which undertakes route planning and user-interface assistance to a vehicle driver. Within this scenario are a number of events which change the operating mode of the navigation systems. For example, two vehicles are configured where one is a master and another is a slave. Events received by each vehicle service platform, for example an accident happens between vehicles, requires that the system adapts and changes mode to recover from the event. In a more complex example, the vehicles get separated on the highway (because, say, one of the drivers had to pull over), the master vehicle switches to planning mode and the slave vehicle to convoy. However, if an accident occurs behind the master and in front of the slave vehicle, meaning only the slave needs to detour it must somehow re-join the master vehicle route planning. The slave navigation system could firstly change to a detour mode (to avoid the accident), then switch to planning mode (to reach a point in range of the master vehicle), and finally switch to convoy mode when close enough the master vehicle. We now explore how the mode specifications and scenarios are formed for the example in UML2. An initial configuration view of the architecture components is illustrated in Figure 3, where an in-vehicle orchestrator component uses a reasoner component to discover local and remote services for vehicle operations.

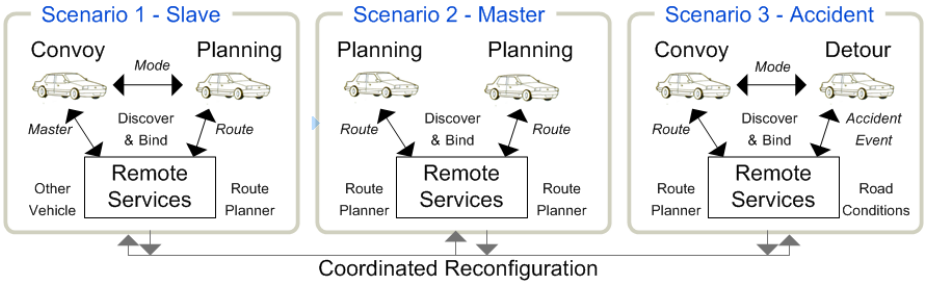

Fig. 2. Mode Scenarios for Vehicle and Remote Service Collaboration

4 Describing Modes in UML2

Our approach to modelling modes in UML2 considers a number of aspects of the language, which we identified from previous work on Darwin+Modes. We utilise the work reported in [5] which proposes modes for the Darwin [10] component

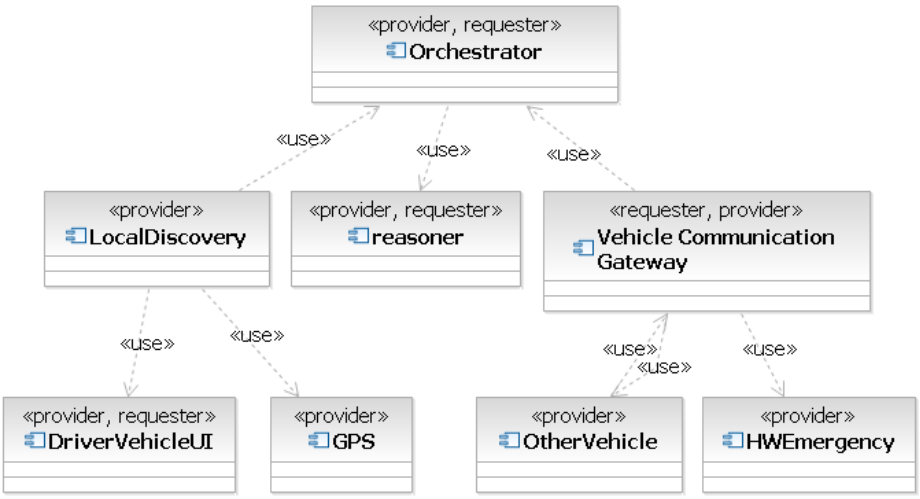

Fig. 3. Architecture for Vehicle Service Components

architecture notation. Firstly, regular component and composite component diagrams are used to describe the set of services in the architecture for the mode domain, detailing the range of services involved in a particular (sub)system task, and their configuration relationships respectively. Secondly, the behaviour (coordination specification) of the architectural changes are described in one or more sequence charts representing scenarios of component behaviour depending on the mode state of the current system architecture configuration. Thirdly, the use of constraints, as part of describing the necessary evaluation and conditions for binding are used to describe architectural correctness. Table 1 lists the mapping for Mode elements, to Darwin and to UML2. Note that those mode elements which are highlighted with a * are proposed enhancements to Darwin+Modes. In addition, Darwin does not describe a process or policy for runtime management of service interactions, although some characteristics of constraints (such as conditional service bindings) could be considered partial runtime management responsibilities.

4.1 Components and Architecture

A high-level architecture configuration is given in UML2 to represent the component specifications and their relationships. An example architecture is illustrated in Figure 3 for a Composite Vehicle Component consisting of a Orchestrator, Reasoner, LocalDiscovery and VehicleCommGateway. Each component will offer services to its clients, each such service is a component contract. A component specification defines a contract between clients requiring services, and implementers providing services. The contract is made up of two parts. The static part, or usage contract, specifies what clients must know in order to use provided services. The usage contract is defined by interfaces provided by a component, and required interfaces that specify what the component needs in

ModeElem.	*Darwin + Modes*	*UML2Objects*	*UML2Notations*	*SoCUsage*
Components	Component	Component	Component	Service Realization
Contract	Ports + Interface	Ports, Interfaces	Composite Structure	Service Specifications (Required/Provided)
Relationship	Bind + Inst	Ports and Connectors	Composite Structure	Service Composition
Constraints	Binding (conditional)	Constraint	OCL, QoS Profile*	Pre- and Post- Conditions (Requests)
Behaviour	Interactions*	Messages and Events	Sequence and Communication	Orchestration and Choreography
Process	Interactions*	Activity and State	Activity and State Machines	Orchestration and Choreography
Policy	Ponder2 Policy*	Behaviour	Policy Profile*	Runtime Policy

Table 1. Mapping Domains of Mode Elements, Darwin, UML2 and SoC Concepts

order to function. The interfaces contain the available operations, their input and output parameters, exceptions they might raise, preconditions that must be met before a client can invoke the operation, and post conditions that clients can expect after invocation. These operations represent features and obligations that constitute a coherent offered or required service. At this level, the components are defined and connected in a static way, or in other words, the view of the component architecture represents a complete description disregarding the necessary state of collaboration for a given goal. Even if the designer wishes to restrict the component diagram to only those components which do collaborate, the necessary behaviour and constraints are not explicit to be able to determine how, in a given situation, the components should interact.

4.2 Contract, Relationship and Constraints

UML2 supports frames (an distinguishable scenario) for which a certain set of components are configured in a specific way to realise the provided and required services in that situation. For example, when an InVehicleServicePlatform service is in "Convoy mode" the components are TaskController, ServiceCoordinator and AnotherVehicle exhibit different roles to carry out the goal of assisting the driver in such a situation. We represent these relationships in UML2 using the composite structure diagram (CSD) notation. UML 2 composite structure diagrams are used to explore run-time instances of interconnected instances collaborating over communications links. The usage contract is specified by the collaboration roles acting as the participants in the collaboration. The types of these roles, often service interfaces, specify what services any service provider playing the role must provide. Connections between the roles indicate interaction between roles and the ports through which they communicate. A collaboration in UML2 may also contain an activity (workflow) or an interaction (sequence) that specifies the realization contract. Note that UML2 provides both a com-

munication diagram and collaboration diagrams. The communication diagram is an extended version of a collaborative diagram from version 1.x and the new collaboration diagram is a composite structure diagram. The later form provides a way to describe components and their roles for a given mode (a form much closer to the meaning of a collaboration scenario).

A constraint in UML2 contains a ValueSpecification that specifies additional semantics for one or more elements. A constraint is a condition (a Boolean expression) that restricts the extension of the associated element beyond what is imposed by the other language constructs applied to that element. Constraint contains an optional name, although they are commonly unnamed. Certain kinds of constraints (such as an association constraint) are predefined in UML, others may be user-defined. One predefined language for writing architectural constraints is the Object Constraint Language (OCL) [12]. OCL is a formal language used to describe expressions on UML models. They are used to express invariant conditions that must be held for the system being modeled. These are described as pre- and post-conditions of the evaluation.

5 A Modes Profile

Using the concepts of UML2 and Modes discussed in sections 2 and 4 respectively, we present here a UML2 Modes Profile (listed in table 2) which assists Service Engineers in identifying mode collaborations, organised hierarchically using Mode Packages. Note that this profile extends and depends on the SENSORIA UML for SoC profile, described in [7], for common service stereotypes (such as service, provider, requester etc).

Stereotype	*BaseClass*	*Contraints*	*Usage*
Mode-Package	Model,Package	If ModeDefault applied, then default mode	A Service Mode Model or Package
Mode-Binding	Connector	Linked between Requesters and Providers	Service Interface and Port Bindings
Mode-Collaboration	Collaboration	in ModePackage	A set of ModeInteractions or ModeActivities
Mode-Interaction	Interaction	in Mode Collaboration	The message behaviour sequences
Mode-Activity	Activity	in Mode Collaboration	Process of mode changes
Mode-Constraint	Constraint	QoS Required or QoS Provided	A Mode Constraint (QoS, Mode Change etc)

Table 2. Modes Profile for UML2

5.1 Models and ModePackage

A UML2 model is a package representing a (hierarchical) set of elements that together describe the physical system being modeled. We define a ModePackage

stereotype as an extension of this definition with a stereotype to designate that a package is being described for a system mode configuration. A ModePackage is expected to have one ModePolicy and one or more elements of type ModeCollaboration, ModeActivity, Events and Signals. A UML2 Model (for Modes) may contain more than one nested package with the stereotype of ModePackage, to provide a set of modes that the system may accommodate. As standard, a UML2 package can import other packages, individual elements from these packages or all their member elements. Packages may also be merged, which also serves a feature of combining modes together. In the case study described in section 3.1 , we would stereotype each vehicle mode as a "ModePackage". At this level of the model, a ModeActivity element in this package serves to describe the ModeCollaboration transitions within the parent mode package.

5.2 ModeCollaboration and ModeBinding

A ModeCollaboration is a UML2 Collaboration containing a Composite Structure Diagram (CSD) to represent the collaborating service components relationships in this mode, one or more ModeInteraction diagrams (sequence and communication diagrams describing the expected message behaviour between the service components) for this mode, and one or more ModeActivity diagrams to represent the higher level ordering of the service interaction behaviour described in the ModeInteraction diagrams. At this level of the package, the ModeActivity serves as a higher-level Message Sequence Chart (hMSC) and the Interaction Diagrams as Basic Message Sequence Charts (bMSCs). Within each CSD, the UML2 element of Connector is stereotyped as a ModeBinding. A Connector represents a (mode) relationship between service components in the collaboration architecture. Also within each CSD, a connector represents a physical link between service component ports, which also contain one or more service interfaces (both required and provided types). Thus, a ModeBinding represents a required connection (or instantiation) in order to carry out the mode behaviour as described in a collaboration interaction set.

5.3 ModeInteraction and ModeActivity

A ModeInteraction contains a single interaction (sequence) diagram and optionally a single communication diagram. The sequence diagram is a message sequence chart describing the sequence of interactions between service components in the mode collaboration. A communication diagram provides an alternative view of the sequence logic for the mode interactions, in a sense a "bird's eye" view of the way the service components collaborate for a given configuration. A ModeActivity is a UML2 Activity which describes a computational process, or in the domain of SoC, a service composition process. The ModeActivity can be specified at either the model level (as a representation of a process to transition between mode collaborations) or at the ModeCollaboration level, to describe the sequencing of ModeInteractions in a mode collaboration.

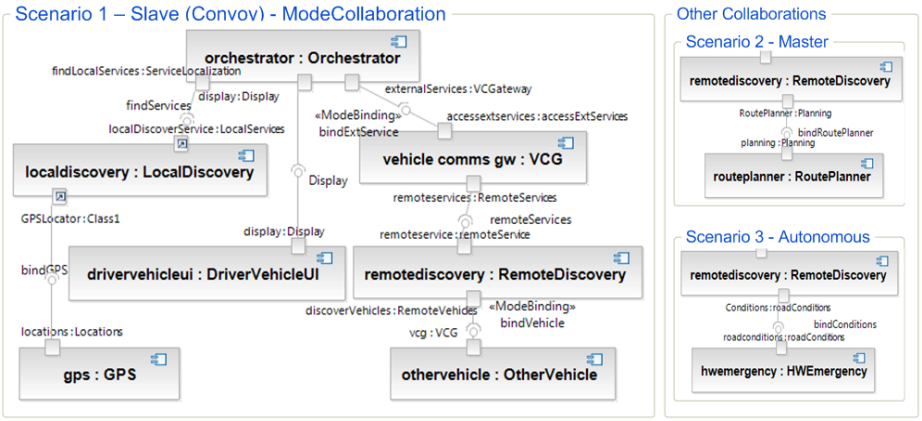

Fig. 4. Composite Structure Diagram (ModeCollaboration) for Slave (Convoy) Component Configuration and alternatives for Planning and Detour Modes

5.4 ModeConstraint and ModePolicy

Constraining changes to architectue and service can be achieved in two ways. Firstly, in a ModeCollaboration Composite Structure Diagram specification, a ModeBinding can be constrained with a ModeConstraint, categorised by a further constraint stereotype. For example, in the domain of Quality-Of-Service (QoS) a QoS Profile can be used to describe the required QoS when connecting a particular service partner (of a particular type). The profile used is based upon a recommendation to the Object Management Group (OMG) in [4]. Additionally, architectural constraints may be specified in OCL or another constraint based language. The constraint language adopted becomes an implementation-dependent aspect of analysing models in UML2. An example constraint applied to a ModeCollaboration is illustrated in Figure 5.

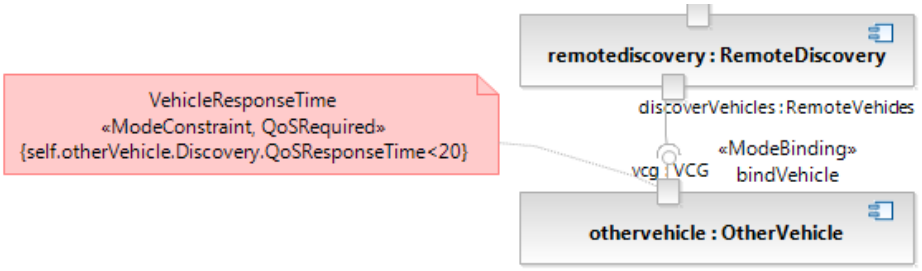

Fig. 5. QoS Constraint for service mode brokering applied to Remote Discovery component and OtherVehicle service

A ModePolicy is a more general specification of the constraints for when mode changes occur at runtime. Whilst the ModeActivity at the model level describes the computational process of a mode change, the policy describes when this change can occur (for a given state of the system). The details of a ModePolicy

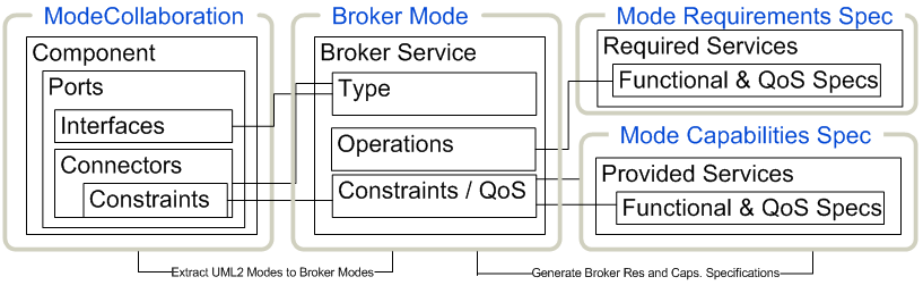

Fig. 6. UML Modes Model to Broker Services and Broker Specifications

are left as future work but can be described using a Distributed Management Policy Language, such as Ponder2.

6 Extracting Modes for Service Brokering

Using the modes model package of mode configurations, exporting the model in the UML XMI Interchange format and using the Dino Broker as an example Brokering implementation, a series of dynamic service brokering requirements and capability specifications can be generated. In this section we detail the steps from a UML mode model to Broker Mode specifications, in preparation for generating broker deployment artifacts. The mapping between model, broker services and broker requirements is illustrated in Figure 6.

6.1 Services, Modes and Constraints

Extracting the capabilities and requirement specifications begins by identifying the different mode packages in the source model. In the case of UML2 each mode package is stereotyped with a 'ModePackage' type, and this builds our first list to reference each package in the model. Next, each mode package is analysed for collaboration elements (stereotyped 'ModeCollaboration'). Each collaboration refers to a series of UML Attributes of type 'uml:Property', which link the components used in a collaboration scenario to the package components defined at a higher level. There is no restriction in UML2 to constrain components to the main model or package level. Any component in a model may be referenced in any other element (at any level). Therefore in building the representation for a Service Broker requirements and capabilities the entire UML model must be available to be referenced for any individual mode package or collaboration. To refine our specification for specific Dino Broker services, we now have three main lists: *a set of mode packages*, *a set of mode collaborations* and *a set of mode components* (services). Further analysis identifies the type of service (required or provided), the operations required or provided, the connectors in the model (port and interface) and constraints. We detail each of these in the following sections.

6.2 Service Components and Connectors

As described in section 2, the Dino broker considers two types of service requirements; 1) required services for a given mode and 2) provided services to announce capabilities that can be matched for a mode of operation. Identifying the type of a service in the mode model relies on alternative properties. Firstly, if the component has required interfaces or is stereotyped as a 'requestor', then the service is marked as a requestor service. Alternatively if the component has a provided interface or is stereotyped as a 'provider' then the service is marked as a provider service. Both requester and provider stereotypes may be applied either by an additional UML profile extension or by adding these keywords as a direct extension to the property of the collaboration. Although we identify requestor and provider types, the Dino requirements and capabilities only consider providers. If a component in the mode collaboration has no interfaces or stereotype, then it is excluded as a service from the broker specifications list.

A connector may or may not exist between a pair of requester and provider type components. To evaluate this we firstly check whether a component has any ports. If ports are located, then for each port a list of connectors for that port is retrieved. The binding between a component refers to an Assembly Connector in UML2. An assembly connector is referenced by a usage, linking the connector, interface and port between components. Therefore, for each connector attached to a port the ends are evaluated against any component in the mode model. Any components located which match the requester component are ignored. This leaves at least one provider component. For each provider component the service operations are extracted as described in section 6.3.

6.3 Service Operations

Building the representation for operations of each type of service requires us to refer to the interfaces of the component in a mode collaboration. Note that operations are not just those which are advertised as offered by the service, but also a list of operations required by a requester component type. Firstly the capabilities of the Dino service are added to the service model in our transformation. If the component has a provided interface, then each operation type, id and name is appended to the Dino Service representation as provided operations. For a provider of operations, building the Dino Service operations is relatively straightforward. However for a requester type service, the connector between service requester and provider instances must be referenced.

6.4 Constraints

We also support extracting ModeConstraints for service bindings, or more specifically a quality of service attribute applied to assembly connectors between two or more components. A ModeConstraint is expected to be expressed in a particular format. For our case study we used the OMG recommendation for a QoS and Fault Tolerance profile in UML [4]. In this example a QoSRequired constraint is

applied to the assembly connector between the remote discovery and other vehicle services. The QoSRequired constraint consists of two key aspects. Firstly, that it has applied the QoSRequired stereotype from the OMG QoS Profile, and secondly that it specifies a Object Constraint Language (OCL) statement which constraints the binding operation. The binding operation, in this case, will be undertaken by the Dino broker. As a final step in building required and provided Dino services, the extract routine traverses the service connectors for any applied constraints. If a constraint is located, then a new QoS attribute is added to the Dino service. The OCL statement is evaluated for object and value expressions, and also added to the Dino service.

6.5 A Modes Browser for Dino

As a visual utility to complement our mechanical extract, we have built a tool which allows the service engineer to navigate through different modes of operation, either by a view of service provision (capabilities of provided services) or by requirements (capabilities of required services). Figure 7 illustrates a view of the browser with the information from the case study. On the left-hand side of the browser, below the specification perspective selection, a list of modes is given. Selecting a mode lists the services in the current perspective of that mode (i.e. all required services). Selecting a service in the list below this, generates a selection of two service deployment artifacts. The first is an .owl document. This document provides a functional description of the service. The functionality of a required or provided service is defined using OWL-S [13], which is an OWL-based ontology for Web services and is an emerging standard for the semantic specification of the functionality of services. An OWL-S description of a service has three parts: profile, process model and grounding. A profile describes the capability of a service operation in terms of its input, output, preconditions and effects; a process model describes details of service operations in terms of atomic, simple and composite processes and a grounding describes details of how to invoke a service operation, and is usually mapped to the service interface description of the service operation. Secondly, a .qos XML document is generated, this provides details of the constraints for service brokering with a service identification, QoS attribute and value required or provided. The Modes Browser is available as part of our service engineering tool suite, known as WS-Engineer [3].

7 Assumptions and Limitations

Our approach is currently limited as we do not consider service mode behaviour, which is required for both Modes analysis and further use of ModeCollaborations for service process synthesis (for example to the BPEL4WS or WS-CDL specifications). Our approach is also dependent on the features of other profiles (such as the UML for service-oriented systems from the SENSORIA project) and the ability to describe particular service profiles so that semantic references can be used in broker service matching. Our approach however is independent

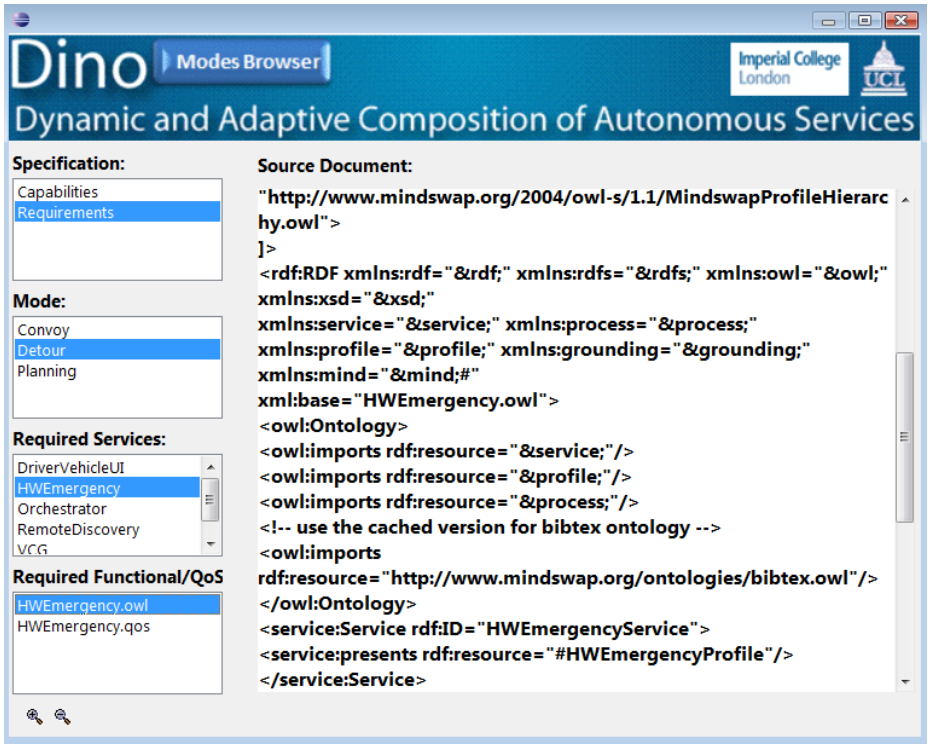

Fig. 7. Dino Service Broker Browser using Vehicle UML Modes Specification

of the actual semantics used for this. We concentrate on allowing the engineer to form the appropriate patterns of service architecture changes and to then generate some useful specifications which can later be used for deployment and execution.

8 Conclusions and Future Work

We believe that the notion of modes helps engineers abstract appropriate elements, behaviour and policy from the services domain, and can facilitate the specification of appropriate control over both architectural change and service behaviour. In this paper we have presented our approach to the modelling of service-oriented computing component architectures using an abstraction of modes to represent the changes in such an architecture. Our future work will explore how these artifacts are more accurately built and analysed, and also on how our approach can assist in the dynamic invocation of services given component requirements and capabilities. We will also detail an approach to generating an implementation layer for switching modes, perhaps through the synthesis of service composition specifications in standards such as BPEL4WS and WS-CDL, representing the tasks for reconfiguration and coordination of services in such an architecture. This work has been partially sponsored by the EU funded project SENSORIA (IST-2005-016004).

References

1. Claudia Baltes, Patrick Koppen, and Paul Muller. Self-management in heterogenous networks using a service-oriented architecture. In *Consumer Communications and Networking Conference, 2007 (CCNC 2007)*, pages 420–424, Las Vegas, NV, USA, 2007. IEEE Computer Society.
2. Vina Ermagan and Ingolf H. Krüger. A uml2 profile for service modeling. In *MoDELS*, Lecture Notes in Computer Science, pages 360–374. Springer, 2007.
3. Howard Foster, Sebastian Uchitel, Jeff Magee, and Jeff Kramer. Ws-engineer:tool support for model-based engineering of web service compositions and choreography. In *IEEE International Conference on Software Engineering (ICSE2006)*, Shanghai, China, 2006. IEEE.
4. Object Management Group. Uml profile for modeling quality of service and fault tolerance characteristics and mechanisms. *Request For Proposal - AD/02-01/07*, 2002.
5. D. Hirsch, J.Kramer, J.Magee, and S. Uchitel. Modes for software architectures. In *Third European Workshop on Software Architecture*. Springer, 2006.
6. Simon Johnston. Uml 2.0 profile for software services. Available at `http://www-128.ibm.com/developerworks/rational/library/05/419_soa`, 2005.
7. Nora Koch, Philip Mayer, Reiko Heckel, Lszl Gnczy, and Carlo Montangero. D1.4b: Uml for service-oriented systems. Technical report, October 2007.
8. Vibhore Kumar, Brian F. Cooper, Greg Eisenhauer, and Karsten Schwan. imanage: Policy-driven self-management for enterprise-scale systems. In Renato Cerqueira and Roy H. Campbell, editors, *Middleware*, volume 4834 of *Lecture Notes in Computer Science*, pages 287–307. Springer, 2007.
9. Ricardo J. Machado, Joao M. Fernandes, Paula Monteiro, and Helena Rodrigues. Transformation of uml models for service-oriented software architectures. In *Proceedings of the 12th IEEE International Conference and Workshops on Engineering of Computer-Based Systems*, pages 173–182, Washington, DC, USA, 2005.
10. J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In W. Schafer and P. Botella, editors, *Proc. 5th European Software Engineering Conf. (ESEC 95)*, volume 989, pages 137–153, Sitges, Spain, 1995. Springer-Verlag, Berlin.
11. Arun Mukhija, Andrew Dingwall-Smith, and David S. Rosenblum. Qos-aware service composition in dino. In *ECOWS '07: Proceedings of the Fifth European Conference on Web Services*, pages 3–12, Washington, DC, USA, 2007. IEEE Computer Society.
12. Object ManagementGroup (OMG). The object constraint language specification 2.0. Available at `http://www.omg.org/docs`, 2007.
13. OWL-S. Owl-based web service ontology. Web Resource, 2007.
14. Peter Van Roy. Self management and the future of software design. In *Formal Aspects of Component Software (FACS '06)*, Prague, Czech Republic, 2006.
15. Ludwig Von Bertalanffy. *General System Theory: Foundations, Development, Applications*. george braziller, New York, NY, 1969.
16. Norbert Wiener. *Cybernetics, or Control and Communication in the Animal and the Machine*. MIT press, Cambridge, MA, 1948.
17. Lawrence Wilkes. Policy driven practices for soa. Available at `http://www.omg.org/news/meetings/workshops/SOA_MDA_WS_Workshop_CD/02-3_Wilkes.pdf`, 2006.