Dino: Dynamic and Adaptive Composition of Autonomous Services*

Arun Mukhija, David S. Rosenblum**, and Andrew Dingwall-Smith

London Software Systems
Department of Computer Science, University College London
Gower Street, London WC1E 6BT, U.K.
{a.mukhija, d.rosenblum, a.dingwall-smith}@cs.ucl.ac.uk

Abstract. Service-oriented computing (SOC) offers a promising solution for dealing with coordination complexity in distributed software systems. Naturally, the infrastructure and technologies for composing services form the backbone of SOC. We argue that SOC has immense potential in enabling collaborations between distributed autonomous services in open dynamic environments, in addition to the restricted business environments that have been the main focus of the work done in SOC so far. We discuss some of the important issues and challenges involved in composing services in open dynamic environments, and give an overview of the Dino approach that we have been developing with an aim to meet these challenges effectively. Dino provides a runtime infrastructure for comprehensively supporting all stages of service composition, namely: service discovery, selection, binding, delivery, monitoring and adaptation. We conclude with a discussion on some of the ongoing and future work on Dino.

1 Introduction

Software systems are already a part of our everyday life, and are destined to become even more pervasive in the coming years. These systems will be increasingly distributed, and operate in dynamic conditions. The high value and efficiency offered by a multitude of software systems comes at a price of high complexity involved in the development and operation of these systems.

The complexity of software systems can be divided into: *computational complexity* and *coordination complexity*. While the efforts to deal with computational complexity have been ongoing for the last several years with considerable success, the focus on managing coordination complexity has been more recent and is mainly driven by the advent of highly distributed software systems.

Service-oriented computing (SOC) offers a promising solution for managing coordination complexity in distributed software systems. SOC builds on the idea of modelling interactions between distributed software components as services

* Partially supported by the EU Integrated Project SENSORIA (IST-2005-016004).
** David Rosenblum holds a Wolfson Research Merit Award from the Royal Society.

White Paper on the Dino approach, 31 January 2007.
http://www.cs.ucl.ac.uk/research/dino/

provided and consumed by the components. The notion of services allows loose coupling between software components, as the services can be described and accessed independently, using standard platform- and implementation language-independent service description languages and communication protocols. The loose coupling in turn allows reusability of software components, implying that a new software system can be developed by composing many of the existing *independently-deployed* and *readily-accessible* software components, while requiring to implement only a minimal number of new application-specific components to be integrated with the existing components. Although loose coupling is not a new concept and is well-recognized within software community as a way to manage coordination complexity, the open XML-based standards proposed as a part of the research on SOC have resulted in their widespread acceptance and have thereby facilitated realization of the loose coupling in practice.

The work on SOC so far has been mostly targeted toward enabling integration of business applications – either within an enterprise (i.e. enterprise application integration) or, more commonly, beyond enterprise boundaries (i.e. B2B integration). However, the potential of SOC is immense in enabling collaborations between autonomous services in open dynamic environments. Service composition in open dynamic environments, though building on the foundational techniques for integration of business applications, is significantly more complex and challenging than the latter. Below we discuss some of the major challenges involved in composing services in open dynamic environments, and argue for the need for comprehensive research in addressing these challenges.

The current applications as well as focus of research on SOC are toward composing services at design time rather than at runtime. Design time service composition is the preferred approach when service partners are known in advance, for example in restricted business environments. However, design time service composition is not feasible in open dynamic environments where services participating in a composition may not be known in advance, i.e. these services can be discovered and composed only at runtime.

Runtime service composition implies that all steps related to the composition process i.e. service discovery, selection, binding and delivery are done at runtime and in an entirely automated manner. This, in turn, imposes additional challenges for *rich* description of services and *intelligent* matchmaking. Moreover, since service partners are not known in advance and are discovered only at runtime, the level of trust between partners is typically low. This calls for appropriate *monitoring* of service agreements, and preferably maintaining a *trust rating* for different service providers.

Another challenge is that the current SOC techniques mostly assume a relatively stable execution environment. That is, once a service composition is formed, the composition is assumed to be largely stable for the period of execution, with only a limited fault-tolerance capability provided to deal with any unforeseen changes in the environment. In open dynamic environments, on the other hand, runtime changes in the execution environment are a norm. These changes might be in the form of changes in the resources available to a service,

which in turn affect the QoS provided by the service or the availability of the service itself. Runtime changes in the execution environment therefore call for an approach for *self-adaptive* service composition able to deal with such changes.

The above challenges have motivated our work on developing an approach for dynamic and adaptive composition of autonomous services, called Dino. In this paper, we give an overview of the Dino approach, and discuss how it helps in meeting the above challenges effectively.

The rest of this paper is organized as follows. In Section 2, we discuss the state of the art in service composition. In Section 3, we provide a detailed description of the Dino approach, including specifications of service requirements and capability, and the runtime infrastructure provided by Dino for enabling dynamic and adaptive composition of autonomous services. Finally, in Section 4, we conclude this paper and discuss future directions of our work on Dino.

2 State of the Art

In the last few years, there has been a significant amount of work done in the area of service composition, mostly for composing Web services. The existing work on service composition broadly covers two main areas: (1) specification languages for describing service compositions, and (2) infrastructure for composing services. Below, we give an overview of the work done in these two areas.

Specification languages for describing service compositions: Most of the ongoing work on service composition, and in particular the standardization effort so far, has been on providing specification languages for describing service compositions. These languages are used for modelling a service composition as a workflow of service interactions.

The interactions between service partners in a service composition can be described either as an *orchestration* or a *choreography*. An orchestration describes a local view of a service composition from the perspective of one service partner and its interactions with other partners. A choreography, on the other hand, describes a global view of a service composition comprising of all service partners and messages exchanged between these partners.

A choreography-based approach is, in effect, a *top-down* approach where a choreography is designed and analysed first, before it's executed by the corresponding service partners. An orchestration-based approach, on the other hand, follows a *bottom-up* approach where each orchestration is designed and implemented individually, while the resulting choreography is formed automatically at runtime by the executions of these orchestrations. The main advantage of a choreography-based approach is that it allows *centralised* analysis of a service composition prior to its execution, which is not possible in orchestration-based approaches where a choreography is formed on-the-fly.

However, a choreography-based approach presumes that all services participating in a composition are known in advance. This is possible only if the

concrete providers for the participating services are also known in advance, because different providers providing the same type of service might have different service requirements. That is, inclusion of different providers for a given service might result in different compositions. Therefore, a global view of a service composition prior to its execution requires the knowledge of concrete service partners forming the composition. This may not be possible in open dynamic environments, where concrete service partners can be discovered and composed only at runtime, rendering choreography-based approaches inappropriate for such environments. Orchestration-based approaches are better suited for open dynamic environments, as here each service partner needs to model only its own interactions with other services, and this can be done at an abstract level without knowledge of the concrete service providers.

On the other hand, in order to counter the lack of centralised analysis possibility available to orchestration-based approaches (when compared to choreography-based approaches), stronger *distributed* analysis and runtime enforcement techniques need to be developed for these approaches.

An emerging standard for describing a choreography is WS-CDL (Web Services Choreography Description Language) [12], and that for describing orchestrations is BPEL (Business Process Execution Language for Web Services) [2]. Both WS-CDL and BPEL are XML-based languages. A service orchestration described in BPEL is like a flow-chart, specifying control logic and flows of interactions between the host service partner (on whose behalf the orchestration is being described) and other partners involved in the orchestration. Since WS-CDL and BPEL have managed to emerge as de-facto standards for describing choreographies and orchestrations respectively, we do not discuss other (mostly similar in principle) service composition languages in this paper.

Infrastructure for composing services: The issue of providing infrastructure for composing services has received relatively less attention, as compared to the research on providing specification languages for describing service compositions. A primary reason behind relatively less work in this area has been an implicit assumption that once a service composition is described clearly, the infrastructure simply needs to support the execution of the defined composition. A popular example of infrastructural support for service composition is the implementation of execution engines for service orchestrations defined in BPEL, such as ActiveBPEL [1]. Such engines are responsible for simply carrying out the execution of an orchestration process defined in BPEL, i.e. handling service requests from other partners and invoking operations on other partners etc.

The role of a service composition infrastructure is much more important in open dynamic environments, as compared to its role in restricted business environments. BPEL has been originally designed, and mostly used, in the restricted environments where service partners are known in advance, and service requirements and capability are not likely to change during execution. Therefore, BPEL engines allow discovery and binding of partners either at design time or the latest

at deployment time. That is, once a binding is established, it cannot be changed at runtime.

An extension to the BPEL infrastructure to allow discovery and invocation of services at runtime, based on the semantic specification of services, has been proposed in [6]. However, with this approach, a service partner is discovered every time an operation is to be invoked, resulting in considerable performance overhead. Moreover, this approach is an extension to BPEL, while our attempt in this work is to provide a general infrastructure for service composition independent of the implementation language of services.

Sycara et al. [11] propose a broker-based approach where brokers are delegated the task of service composition. This approach is closest to our approach, as our approach also relies on the concept of brokers for service compositions. Even though we share the same aims, our approach differs from the above approach on several fundamental issues. For instance, our approach requires formal description of service requirements, in addition to the description of service capability, to allow reliable automated matchmaking. In addition, our approach focuses on explicit and detailed specification of QoS in the description of service requirements and capability, as the QoS plays an important role in selecting a service provider. One of the novel features of our approach is the comprehensive support for automated adaptation of a service composition, which involves monitoring the QoS being delivered and detecting any violation of the service agreement established between service partners.

Yu et al. [14] also propose a broker-based approach for selecting and composing services based on their QoS. However, this approach assumes a choreography view of the service composition to be available, and aims to identify optimum service partners to be composed together in accordance with this choreography view. Clearly, such an approach is not appropriate for the open dynamic environments that we have been targeting.

3 The Dino Approach

W3C defines a service as "an abstract resource that represents a capability of performing tasks that form a coherent functionality from the point of view of providers entities and requesters entities. To be used, a service must be realized by a concrete provider agent." [13].

In the SOC paradigm, distributed software components interact with each other by providing services to other components and consuming services provided by other components. A software component providing a certain service is called a service provider (SP), and a software component interested in consuming a service provided by another component is called a service requestor (SR). SR and SP are just the logical roles played by a software component. In fact, the same software component can – and, in fact, is likely to – play the role of a SP as well as SR in a given composition. That is, a component might provide certain services, and at the same time require some services from some other components. Services are provided and consumed by way of message exchanges between a SR and a

SP. We will simply refer to a software component providing and/or requiring a service as a *service entity* when the role played by the component is not relevant.

The service composition approach of Dino builds on the idea of *rich* specification of service requirements and capability to allow automated discovery and selection of services. Dino provides a runtime infrastructure for service composition. The runtime infrastructure consists of a number of Dino brokers, which are responsible for service discovery, selection, binding, delivery, monitoring and adaptation. Detailed description of the various stages of service composition in Dino is given below.

First, we introduce the concept of *modes* of a service entity. A service entity might have alternative modes of operation, only one of which is active at a given time. A change in mode is usually triggered by a change in the execution environment of the service entity, such as a change in the resources available or a change in user's needs / preferences. A change in mode implies a change in the internal configuration of a service entity. From an external perspective, a change in mode might mean a change in service requirements or capability of the service entity. Hirsch et al. [3] provide an architectural approach, based on the Darwin architecture description language [5], for modelling different modes of a software component in terms of differences in their required and provided interfaces. We build on the foundational work by Hirsch et al. for modelling modes of a service entity at the architecture level, and provide support for the different modes of operation during service composition.

To illustrate the design and operation of Dino, we take an example from the automotive domain, which is adopted from the case study used by Hirsch et al. [3] for explaining the concept of modes. A service entity called Driving-Assistant provides a route planning service (RPS) for providing route-guiding instructions to a driver. Driving-Assistant, in turn, requires some other services to be able to provide the route planning service. The services required by Driving-Assistant depend upon its current mode of operation. That is, a change in mode might imply a change in the service requirements of Driving-Assistant.

3.1 Service composition stages in Dino

3.1.1 Specification of service requirements and capability: Specifications of service requirements and capability play a very important role in the automated composition of services. Ideally, these specifications should be as unambiguous and comprehensive as possible. A simple syntactic specification of a service interface, such as the one described using industry standard WSDL (Web Services Description Language), does not offer a completely unambiguous solution. Recent work on semantic specification of services, led by OWL-S [8], aims to utilize shared ontologies for avoiding any potential ambiguity in service descriptions.

In Dino, every SR is required to specify its service requirements in an XML-based document called ReqDoc. Similarly, every SP is required to specify its service capability in an XML-based document called CapDoc. Both ReqDoc and

CapDoc rely on OWL-S [8] for specifying service functionality, which is an OWL-based ontology for the semantic specification of services. In addition, ReqDoc and CapDoc provide special constructs for describing the QoS for each required and provided service. Though OWL-S allows extension of its basic specification for describing non-functional properties, we have decided in favour of maintaining the functional and non-functional descriptions separately for easy manageability. In particular, for every service, the functionality of the service is described in a .owl file and the QoS is described in a .qos file. QoS parameters described in the .qos file can refer to the service operations described in the .owl file, if required.

Both ReqDoc and CapDoc are divided into mode segments, with each mode segment specifying services required or provided in the corresponding mode. Figure 1 shows a simplified version of the ReqDoc of Driving-Assistant service entity. Driving-Assistant has three possible modes: autonomous, convoy and detour. In the autonomous mode, the route is planned autonomously by Driving-Assistant, and it requires only a GPS service and Map service. In the convoy mode, the driver needs to follow another vehicle, and therefore Driving-Assistant requires input from the route planning service of another vehicle. And in the detour mode, the vehicle is guided by an external emergency system (to avoid some problems, such as accident or works in the street), and therefore Driving-Assistant requires highway emergency service (HES).

```
<ReqDoc name="Driving-Assistant">
     <mode name="autonomous">
          <service name="GPS" functional="gps-req.owl" qos="gps-req.qos"/>
          <service name="Map" functional="map-req.owl" qos="map-req.qos"/>
     </mode>
     <mode name="convoy">
          <service name="RPS" functional="rps-req.owl" qos="rps-req.qos"/>
     </mode>
     <mode name="detour">
          <service name="HES" functional="hes-req.owl" qos="hes-req.qos"/>
     </mode>
</ReqDoc>
```

Fig. 1. Requirements document of Driving-Assistance

In Figure 1, the .owl file referred by the `functional` attribute of a <`service`> element contains the functional description of the service in OWL-S, and the .qos file referred by the `qos` attribute contains the QoS description of the service. The name of a service is used for internal reference only. Format of CapDoc is similar to that of ReqDoc. However, the functional description in ReqDoc is more abstract than the one in CapDoc, as explained below.

OWL-S description of a service consists of three parts: profile, process model and grounding. Profile describes operations of a service complete with information about their input messages, output messages, preconditions and effects. Process model describes in detail what the service does in terms of atomic processes that can be executed directly and composite processes that are formed by

combining other atomic and/or composite processes. However, composite process descriptions are not relevant to the working of Dino, as these are used for describing composite services (similar to BPEL) and are not used in the working of Dino. Finally, grounding describes details of how to access the service including details of communication protocols, serialization techniques etc. For more details on OWL-S, please refer to [8].

The .owl file in ReqDoc specifies the OWL-S profile of a required service, while the .owl file in CapDoc contains at least the grounding information in addition to the OWL-S profile for a provided service. The advantage of OWL-S is that the descriptions of the required and provided services can refer to the shared ontologies for the common understanding of the terms used in the descriptions. This helps in removing any ambiguity and allows for automated matchmaking. Another advantage offered by OWL-S is that services can be described following the principle of design-by-contract by including preconditions and effects for every operation, as opposed to a simple interface specification that can be described using WSDL.

Below we give a brief overview of the QoS specifications in a .qos file. Similar to the .owl file, the .qos file also contains pointers to the shared ontologies for the common understanding of the terms used in the QoS specification.

We assume that all QoS parameters are quantifiable. Examples of QoS parameters include domain-independent parameters like response time, throughput, availability, cost, location (even though cost and location are not actually 'quality' parameters, these are usually correlated with other quality parameters), and domain-specific parameters like accuracy of results, fidelity of data etc. Each QoS parameter is specified in a <`qos`> element. Attributes of the <`qos`> element are: `name` (specifying the unique name of a QoS parameter), `operation` (reference to the service operation(s) in the .owl file to which this parameter corresponds; if no operations are specified then this parameter corresponds to all operations specified in the .owl file), `unit` (unit of measurement), `minVal` (for numerically quantifiable values only; this is the lowest value of an offered QoS and it appears in CapDoc only), `maxVal` (opposite of minVal, i.e. the highest value of an offered QoS), `mpVal` (for numerically quantifiable values only; this is the most preferred value of a required QoS and it appears in ReqDoc only), `lpVal` (opposite of mpVal, i.e. the least preferred value of a required QoS), `enum` (enumeration of discrete values; the order of enumeration is from the most preferred to least preferred when appearing in ReqDoc, while in CapDoc the order is not important), `confidence` (confidence level of the service provider in these values), `priority` (relative priority of this parameter compared to other parameters; it appears in ReqDoc only). Many of the above attributes are optional in a QoS specification. An example specification for an offered QoS described in CapDoc is shown in Figure 2.

A number of <`qos`> elements can be grouped together within an <`and`> or <`alt`> element. An <`and`> element indicates a conjunction, i.e. all elements enclosed within an <`and`> element hold together. An <`alt`> element, on the other hand, indicates an exclusive disjunction, i.e. only one of the elements enclosed

```
<qos name="responseTime" operation="id1" unit="ms"
     minVal="50" maxVal="250" confidence="0.99"/>
```

Fig. 2. QoS specification for an offered service

within an <alt> element holds. Either of these elements may contain other <and> or <alt> elements in addition to the <qos> elements. An <alt> element can be used, for example, by a SP to indicate different alternative QoS values it can offer, probably at different prices. An <and> element can be used for grouping several <qos> elements together within an <alt> element, to present these elements collectively as an atomic alternative.

The <and> and <alt> elements together allow specifying complex QoS policies of service entities. These policies are used by Dino brokers for matchmaking.

We assume that detailed description of the operations required and provided by service entities is sufficient for automated matchmaking. The preconditions and effects described in the OWL-S specifications of operations provide implicit constraints on the order of invocation of these operations. The conversation logic, i.e. what messages to exchange using these operations, is implemented within the service entities involved in the exchange. The conversation logic can be implemented in a conventional programming language or a special-purpose service orchestration language like BPEL. However, if the conversation logic is implemented in BPEL, the conventional use of BPEL needs to be modified to take advantage of the Dino infrastructure instead of simply using a BPEL execution engine. Next, we discuss the service discovery and selection in Dino.

3.1.2 Service discovery and selection: Once service entities have specified their service requirements and capabilities in the respective ReqDoc and CapDoc documents, they can utilize the Dino runtime infrastructure for collaborating with other service entities by forming a service composition.

The process of service discovery is always initiated by a SR. A SR invokes a Dino broker, and passes its ReqDoc to the broker. Additionally, the SR provides the broker with an ordered list of modes that it is willing to accept. The list is ordered according to the preference of the SR. It is possible that the SR provides only a single mode that it is willing to accept. Providing a list of modes gives more flexibility to the Dino broker such that if the most preferred mode of the SR cannot be accepted due to unavailability of the required services, then the Dino broker may attempt to satisfy the requirements for the next mode and so on.

A Dino broker can be located anywhere in the network, e.g. on the same node as a SR or on a trusted third party node. However, as discussed later, for the better monitorability reasons, it is recommended that a Dino broker be hosted on the same node as a SR interested in using the service of the broker.

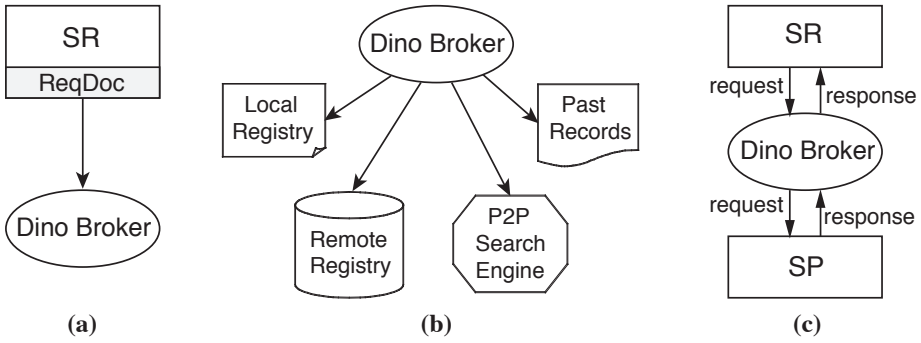

Fig. 3. Working of Dino broker

Upon invocation, the Dino broker begins search for the services that can satisfy the service requirements specified in the ReqDoc passed to it. A search for services is done using a number of different mechanisms, which are provided in Dino with an aim to widen the chances for a positive search result. The search mechanisms include searching a service registry such as UDDI extended with OWL-S [10], local registry of Dino brokers where services can register, a search engine such as a peer-to-peer search engine for services developed as a part of the Dino project, or simply past records maintained by Dino brokers containing the previous search results. Figure 3(a) shows a SR registering its ReqDoc with a Dino broker, and Figure 3(b) shows the Dino broker searching for the required services.

Although a CapDoc contains information on services that a SP can provide in all its different modes, the published capability of the SP (i.e. the one published on a service registry, or the one made searchable by a search engine) at a given time contains information only on those services that the SP is currently able to provide.

The search for candidate services is done by matching the functionality and QoS. The functional matching is done by matching the OWL-S profiles described in service requirements with the OWL-S specifications in service capability descriptions. For the functional matching, a number of matchmakers for OWL-S specifications are available. We have used the OWL-S API developed at MINDSWAP [7] for developing a matchmaker for functional matching. For matching the QoS, we use our own matching algorithm. When more than one service match a service requirement, the Dino broker uses a QoS-based selection algorithm for selecting the best match.

Once the best match for each of the required services is selected, the Dino broker establishes service agreements with the selected SPs.

3.1.3 Service delivery and monitoring: Once all the required services are selected and the corresponding service agreements have been established, the delivery of services from the selected SPs to the SR can begin. As mentioned

earlier, a service is delivered by exchange of messages between a SR and a SP. Currently, we consider only point-to-point RPC style interactions between service entities for service delivery. However, in future we plan to provide support for other interaction styles, such as publish-subscribe and shared tuple space, that might be useful for group communication in some service-oriented applications.

In the RPC style interactions in Dino, the SR sends a service invocation message to the Dino broker. The Dino broker then forwards this message to the corresponding SP. Similarly, the response from the SP is sent to the broker, which then forwards this response to the SR. Figure 3(c) shows service interactions between the SR and SP. SR and SP do not have a direct reference to each other, and interact only through Dino broker. This is useful in case an adaptation of the service composition is required, as discussed in the next subsection. However, Dino provides flexibility to the SR to opt for a direct delivery of service (i.e. without the involvement of Dino broker), if required, for example due to hard real-time constraints on the response time etc. If direct delivery is desired, then the <`service`> element in the ReqDoc contains a special attribute called `direct-delivery` with a value of `yes`. In such a case, the Dino broker simply provides a reference of the SP to the SR at the end of the service selection phase, and the SR and SP interact with each other directly thereafter. However, the default method of delivery is through a Dino broker.

When a service is delivered through a Dino broker, the broker is able to perform syntactic mapping of the messages exchanged between the SR and SP, when these messages have matching semantics but differing syntax, as explained next. Recall that the matchmaking is performed on the basis of semantic specifications of service requirements and capability. However, there might be syntactic incompatibilities between the SR and SP, e.g. they might use different names for an operation while assuming the same semantics for the operation. To avoid this problem, the SR always sends messages using the same nomenclature as in the profile used to specify service requirements in ReqDoc. The Dino broker internally translates these messages into the nomenclature expected by the SP, using the information available from the SP's grounding information. The response returned by the SP is similarly translated back by the Dino broker before being forwarded to the SR. This translation is facilitated by the OWL-S API [7].

Apart from the syntactic mapping, Dino also provides support for translating messages across heterogeneous communication protocols. This implies that the SR and SP might use otherwise-incompatible communication protocols, but are still able to communicate with each other. This kind of translation is made possible by integrating Dino broker with an underlying technology called Service Bridge, which is developed as a part of the Dino project. As the name indicates, a Service Bridge is able to translate messages dynamically across heterogeneous communication protocols. More details on the Service Bridge can be found in [9].

Delivery of messages is carried out in a transactional manner by Dino brokers. A failed delivery is retried several times (with the actual number of retries

dependent upon the current network conditions) to ensure reliable delivery of messages. In future, we intend to provide support for secured delivery of messages using the Dino runtime infrastructure.

In addition to supporting delivery of service messages, Dino brokers play an important role in monitoring the service agreements. This involves mainly monitoring the QoS being delivered, and detecting any violation of the service agreement. The actual monitoring is performed by collaboration between the SR and Dino broker. This is because only a few QoS parameters can be monitored independently by the Dino broker (such as availability), while others (such as accuracy of results) can be monitored by the SR only. For some parameters, such as response time, even though the precise values can be obtained when monitored by the SR, reasonably approximate values can be obtained when monitoring is done by the Dino broker, if the broker and the SR are located on the same network node. Our aim is to maximise the proportion of monitoring done by the Dino broker, to relieve the SR from the overhead involved in monitoring. For the parameters that can be monitored by the SR only, the SR provides feedback to the Dino broker in case any violation of the service agreement is detected.

Next, we discuss the adaptation actions performed by the Dino broker in response to any violation of the service agreement.

3.1.4 Service recomposition: When a Dino broker discovers a violation of a service agreement (either by monitoring on its own, or by getting feedback from the SR), it initiates adaptation actions. At this stage, we are not concerned with the actual cause behind a violation – be it a hardware failure, mobility of a node, or even malicious behaviour of a SP. The adaptation involves selecting an alternative SP to replace the current SP.

If more than one SP had matched successfully during the initial search, the Dino broker need not conduct a new search, and tries to establish a service agreement with a SP that had previously matched successfully. Otherwise, the Dino broker needs to search for an alternative SP, in a similar manner as done initially. The criteria for selecting a SP from among several candidates remains the same as during the initial search. It is possible that no alternative SP is found for a required service. In this case, the SR can either inform a new mode or lower its criteria for the selection of the required services in the current mode.

If an alternative SP is found successfully and a new service agreement is established, the Dino broker is responsible for managing the handover from the old SP to the new SP, ideally in a way transparent to the SR. The handover typically involves transferring the state of the old SP to the new SP, so that the new SP is able to resume the execution correctly. For the state transfer, we argue for a change in the conventional way of maintaining the persistent state in a service entity (i.e. the state that needs to remain persistent in between invocations). In the conventional way, the state of a software component is its internal matter and is hidden from the external world. Even though some parameters of the state can be queried, the structure of the state is largely oblivious to other components. We argue that increasingly (and in particular in SOC), a SR might

be interested in switching its SP at runtime, especially for long running interactions. At the same time, the SR would not like to loose the results of its past computations, i.e. its state maintained by the SP. One option will be for the SR to maintain a copy of the state relevant to it at all times. But this is likely to create unnecessary overhead for the SR, and the SR may not even understand the semantics of the state.

This is similar to a real world situation where a patient (i.e. SR) wants to switch her family physician (i.e. SP), for example due to relocation. But at the same time, the patient would not want to loose her past medical history records, as these are very important for the patient. The common solution in this case is for the patient's current family physician to send her records to her new family physician. These records are already in the form that any qualified physician will be able to understand and make sense of. The same model of transfer of records can be replicated in dynamic service recomposition, with the state of the old SP being transferred to the new SP. This transfer can take place with the Dino broker acting as an intermediary. A fundamental problem with this scheme, however, is that the new SP should be able to understand and make sense of the state being transferred to it, in order to use the state properly.

The only solution to the above problem is for all the SPs providing similar functionality, i.e. the ones that can potentially replace each other at runtime, to have a common shared understanding of the persistent state to be maintained. Fortunately, this solution is realizable using the XML schemas, similar to the ones referred by WSDL and SOAP for providing information on the data structure of messages. In particular, we argue that for the 'transferable' state (i.e. the state that needs to be transferred from one SP to another, in case a SR wants to switch the providers at runtime), the mutually-replaceable SPs should conform to a shared XML schema of the state. Conforming to a shared XML schema implies ability of a SP to expose its state according to the XML schema in case the SP is to be replaced by another SP, as well as the ability of a SP to load a state specified according to the XML schema in case the SP is replacing another SP. We are currently working on formalizing the details of such a state transfer.

However, in case of an abrupt loss of a service, the state transfer between SPs may not be possible. The concerned service entities need to implement appropriate recovery mechanisms for such a case.

Once the new SP is ready, the service requests from the SR are forwarded to the new SP instead of the old SP, even though the SR itself might be oblivious to a change in the actual SP. During the time that the new SP is selected and activated, any service requests from the SR are queued within the Dino broker, to be forwarded to the new SP once the new SP is ready to accept requests.

A runtime change in the mode of a SR or SP might result in changes in the corresponding service requirements or capability. The concerned Dino brokers are responsible for carrying out recomposition of services accordingly to accommodate these changes. In particular, a change in the service capability might result in the violation of a service agreement, and thus trigger adaptation of the service composition as described above. Whereas, a change in the service re-

quirements results in the invocation of a Dino broker to discover and select new SPs for the newly required services, in a similar manner as done initially, while bindings with any obsolete (no longer required) services can be closed safely.

The dynamic service recomposition ability allows a Dino-managed service-oriented application to continue its execution unhindered even in the wake of unpredictable changes in its execution environment.

3.2 Implementation of the Dino runtime infrastructure

A prototype Dino broker is implemented in Java, and is accessible as a SOAP Web service. A partial specification of the `DinoBroker` interface (showing basic functions) is shown in Figure 4. A WSDL description is generated from the `DinoBroker` interface for SOAP access.

```
public interface DinoBroker {
    public String startSession();
    public void quitSession(String sessionId);
    public void registerReqDoc(String sessionId, String reqDocURL);
    public SelectedModeResponse selectMode(String sessionId,
                        String[] requestedModes);
    public Param[] invokeService(String sessionId, String serviceName,
                        Param[] params);
}
```

Fig. 4. The Java interface to the Dino broker

Continuing with the automotive case study used earlier on in the description, we take another example scenario from this domain. A mobile client (in a moving vehicle) is interested in performing location-sensitive search for restaurants. There are two modes in which the client can operate. The `user-input-location` mode requires only the `restaurant-search` service. This service takes a number of inputs, including the location, which are used to search for restaurants. In this mode, the client needs to ask the user for the current location. In the `gps-location` mode, two additional services are required. The `gps` service returns the latitude and longitude at which the client is located. This will be a local service, possibly an OSGi component which accesses a GPS device. The `latlong-to-city` service takes a latitude and longitude, and returns the town or city which best corresponds to the location. Generally, the client will ask for `gps-location` as the preferred mode and `user-input-location` as an alternative mode.

To start using the Dino broker, a client must call the `startSession` method. This method returns a session identifier which must be used for all subsequent communication with the Dino broker. As multiple clients can use the Dino broker simultaneously, and the Dino broker maintains state for each client, session identification is necessary to match client calls to existing session state.

Having obtained a session, the client can register a ReqDoc using the `registerReqDoc` method. This requires a URL to be provided from which the Dino broker can retrieve the ReqDoc. Each session has a single ReqDoc, and calling this method again results in an existing ReqDoc being replaced by the new ReqDoc.

Once a ReqDoc has been registered by a client, it must select which mode it wants to use. This will determine what services the Dino broker has to discover. The client provides an array of possible modes that it is willing to accept, in the order of preference. The Dino broker attempts to satisfy the requirements of one of the modes and, if successful, reports which mode it has selected inside a `SelectedModeResponse` object. If no mode can be satisfied, a `ServiceDiscoveryException` is thrown. Once a mode has been selected, the client can invoke services which are required in the selected mode, using the `invokeService` method. The parameters which are passed to the service and returned by it are OWL-S parameters which are translated by the Dino broker, using the OWL-S API [7] and the OWL-S description of the invoked service, into the form understood by the invoked service.

In the restaurant search scenario, the `gps` service is invoked with no parameters and returns a parameter representing latitude and longitude. This output in then used as input to the `latlong-to-city` service which requires latitude and longitude as input and returns the name of a city. This city name is then used as one of the input parameters to the `restaurant-search` service.

4 Conclusion and Future Work

In this paper, we have identified challenges involved in composing distributed autonomous services in open dynamic environments, and introduced the Dino approach to meet these challenges effectively. Dino allows dynamic composition of autonomous services by formalizing functional and non-functional specifications of service requirements and capabilities, and providing an infrastructure for composing services at runtime. The runtime infrastructure provided by Dino consists of a number of Dino brokers, which are responsible for discovery, selection, binding and delivery of services. In addition, the Dino brokers are responsible for monitoring the services being delivered and taking adaptation actions in response to any violation of a service agreement or a change in service requirements, thereby enabling self-adaptive service compositions.

Although a prototype implementation has been built, Dino is still in its evolutionary phase. Below we discuss some of the work that is ongoing or is planned for near future.

We are currently working on supporting multiple interaction styles, such as publish-subscribe and shared tuple space, in addition to the RPC style interactions in Dino. We are also in the process of formalizing details of the runtime state transfer during service recomposition, and evaluating the state transfer approach by applying it to a number of case studies.

We will continue our work on developing the Dino prototype system for evaluating current as well as future design features of Dino. We plan to deploy the prototype in a real-world setting and carry out detailed evaluation, in large part through our work in the EU-funded project SENSORIA.

In Dino, the service requirements and capability specifications provided by individual service entities act as the validity constraints for a service composition formed by combining these entities. The runtime infrastructure provided by Dino is responsible for ensuring that these validity constraints are satisfied at all times. Correctness and completeness of these validity constraints is, therefore, important in this respect. We are currently working on ways to analyse and verify the correctness and completeness of the service requirements and capability specifications. We plan to build tools for the automated generation of service requirements and capability specifications from the architectural descriptions of the service entities, in addition to the verification of these specifications. With the help of these tools, we intend to address the need for distributed analysis in service orchestrations.

References

1. ActiveBPEL Open Source Engine. http://www.activebpel.org/
2. *Business Process Execution Language for Web Services Version 1.1*, May 2003. http://www.ibm.com/developerworks/library/ws-bpel/
3. D. Hirsch, J. Kramer, J. Magee and S. Uchitel, "Modes for Software Architectures", *Proc. of 3rd European Workshop on Software Architecture (EWSA'06)*, Sept. 2006.
4. Knopflerfish – Open Source OSGi. http://www.knopflerfish.org/
5. J. Magee, N. Dulay, S. Eisenbach and J. Kramer, "Specifying Distributed Software Architectures", *Proc. of 5th European Software Engineering Conference (ESEC'95)*, Sept. 1995.
6. D.J. Mandell and S.A. McIlraith, "Adapting BPEL4WS for the Semantic Web: The Bottom-Up Approach to Web Service Interoperation", *Proc. of 2nd International Semantic Web Conference (ISWC'03)*, Oct. 2003.
7. Maryland Information and Network Dynamics Lab Semantic Web Agents Project, OWL-S API. http://www.mindswap.org/2004/owl-s/api/
8. OWL-S Ontology Web Language for Services. http://www.daml.org/services/owl-s
9. W. Perry, S. Uchitel, D.S. Rosenblum and A. Mukhija, *Service-Oriented Middleware Services*, Deliverable D6.3a, EU project: SENSORIA, Aug. 2006.
10. N. Srinivasan, M. Paolucci and K. Sycara, "Adding OWL-S to UDDI, Implementation and Throughput", *Proc. of 1st International Workshop on Semantic Web Services and Web Process Composition*, July 2004.
11. K. Sycara, M. Paolucci, J. Soudry and N. Srinivasan, "Dynamic Discovery and Coordination of Agent-Based Semantic Web Services", *IEEE Internet Computing*, Volume 8, Issue 3, May 2004.
12. *Web Services Choreography Description Language Version 1.0*, W3C Candidate Recommendation 9 November 2005. http://www.w3.org/TR/ws-cdl-10/
13. *Web Services Glossary*, W3C Working Group Note 11 February 2004. http://www.w3.org/TR/ws-gloss/
14. T. Yu and K-J. Lin, "A Broker-Based Framework for QoS-Aware Web Service Composition", *Proc. of International Conference on e-Technology, e-Commerce and e-Service (EEE'05)*, March-April 2005.