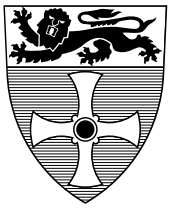
UNIVERSITY OF
NEWCASTLE



University of Newcastle upon Tyne

COMPUTING SCIENCE

Secure and Provable Service Support for Human-Intensive Real-Estate Processes

E. Ribeiro de Mello, S. Parastatidis, P. Reinecke, C. Smith, A. van Moorsel, J. Webber.

TECHNICAL REPORT SERIES

No. CS-TR-960 May, 2006

TECHNICAL REPORT SERIES

No. CS-TR-960 May, 2006

Secure and Provable Service Support for Human-Intensive Real-Estate Processes

Emerson Ribeiro de Mello, Savas Parastatidis, Philipp Reinecke, Chris Smith, Aad van Moorsel, Jim Webber.

Abstract

We introduce SOAR, a service-oriented architecture for the real-estate industry that embeds trust and security, allows for formal correctness proofs of service interactions, and systematically addresses human interaction capabilities through web-based user access to services. We demonstrate the features of SOAR through a Deal- Maker service that helps buyers and sellers semi-automate the various steps in a real-estate transaction. This service is a composed service, with message-based interactions specified in SSDL, the SOAP service description language. The implemented embedded trust and security solution deals with the usual privacy and authorization issues, but also establishes trust in ownership and other claims of participants. We also demonstrate how formal techniques can proof correctness of the service interaction protocol specified in SSDL. From an implementation perspective, a main new contribution is a protocol engine for SSDL. A proof-of-concept demonstration is accessible for try-out

Printed and published by the University of Newcastle upon Tyne,
Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.

Bibliographical details

RIBEIRO DE MELLO, E., PARASTATIDIS, S., REINECKE, P., SMITH, C., VAN MOORSEL, A., WEBBER, J..

Secure and Provable Service Support for Human-Intensive Real-Estate Processes
[By] Emerson Ribeiro de Mello, Savas Parastatidis, Philipp Reinecke, Chris Smith, Aad van Moorsel, Jim Webber.

Newcastle upon Tyne: University of Newcastle upon Tyne: Computing Science, 2006.

(University of Newcastle upon Tyne, Computing Science, Technical Report Series, No. CS-TR-960)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE
Computing Science. Technical Report Series. CS-TR-960

Abstract

We introduce SOAR, a service-oriented architecture for the real-estate industry that embeds trust and security, allows for formal correctness proofs of service interactions, and systematically addresses human interaction capabilities through web-based user access to services. We demonstrate the features of SOAR through a Deal-Maker service that helps buyers and sellers semi-automate the various steps in a real-estate transaction. This service is a composed service, with message-based interactions specified in SSDL, the SOAP service description language. The implemented embedded trust and security solution deals with the usual privacy and authorization issues, but also establishes trust in ownership and other claims of participants. We also demonstrate how formal techniques can proof correctness of the service interaction protocol specified in SSDL. From an implementation perspective, a main new contribution is a protocol engine for SSDL. A proof-of-concept demonstration is accessible for try-out

About the author

Emerson Ribeiro de Mello is with Federal University of Santa Catarina, Departamento de Automacao e Sistemas, Florianopolis, Brasil.

Savas Parastatidis is with Microsoft, Redmond, USA..

Philipp Reinecke is with Humboldt-Universitaet Berlin, Institut fuer Informatik, Berlin, Germany,

Christopher Smith is a PhD student at the University of Newcastle-upon Tyne, and researching in the area of self-managing systems; specifically software architectures and distributed decision-making algorithms.

Aad van Moorsel joined the University of Newcastle in 2004. He has worked in a variety of areas, from performance modelling to systems management, web services and grid computing. Most recently, he was responsible for HP's research in web and grid services, and worked on the software strategy of the company.
His research agenda is in the area of self-managing systems.

Jim Webber is with ThoughtWorks, Sydney, Australia.

Suggested keywords

WEB SERVICES,
SERVICE-ORIENTED COMPUTING,
SSDL,
PI-CALCULUS,
SECURITY

Secure and Provable Service Support for Human-Intensive Real-Estate Processes

Emerson Ribeiro de Mello,* Savas Parastatidis,† Philipp Reinecke,‡
Chris Smith,§ Aad van Moorsel, Jim Webber¶

University of Newcastle
School of Computing Science
Newcastle upon Tyne
United Kingdom

Preface: this is a submission to the IEEE Services Computing Contest 2006. The first nine pages constitute the 'paper', while the overall document (that is, including the appendices) is the 'report'. The third element of our submission is the web site to our real-estate DealMaker service, which is at `http://vs-soc.ncl.ac.uk:8180/CloseTheDeal/index.html` (account: `close`, password: `thedeal` [4]).

Abstract

The real-estate industry is an interesting target for service-oriented computing, for several reasons. The participating parties are extremely diverse and there is a high proportion of human activity and interaction involved in traditional real-estate transactions. This implies that (partial) automation of such processes must be done in highly flexible and trusted manner, with natural inclusion of the human element. The most promising response computer science offers to these challenges is found in service-oriented approaches. In this paper, we argue how service-oriented computing can potentially disrupt the real-estate industry from a business perspective. We introduce SOAR, a service-oriented architecture for the real-estate industry that embeds trust and security, allows for formal correctness proofs of service interactions, and systematically addresses human interaction capabilities through web-based user access to services. We demonstrate the features of SOAR through a DealMaker service that helps buyers and sellers semi-automate the various steps in a real-estate transaction. This service is a composed service, with message-based interactions specified in SSDL, the SOAP service description language. The implemented embedded trust and security solution deals with the usual privacy and authorization issues, but also establishes trust in ownership and other claims of participants. We also demonstrate how formal techniques can proof correctness of the service interaction protocol specified in SSDL. From an implementation perspective, a main new contribution is a protocol engine for SSDL. A proof-of-concept demonstration is accessible for try-out [4].

* Emerson Ribeiro de Mello is with Federal University of Santa Catarina, Departamento de Automacao e Sistemas, Florianopolis, Brasil, `emerson@das.ufsc.br`.

† Savas Parastatidis is with Microsoft, Redmond, USA, `savas@parastatidis.name`.

‡ Philipp Reinecke is with Humboldt-Universität Berlin, Institut für Informatik, Berlin, Germany, `preinecke@informatik.hu-berlin.de`.

§ Chris Smith and Aad van Moorsel are with University of Newcastle, School of Computing Science, Newcastle upon Tyne, UK, `{c.j.smith4,aad.vanmoorsel}@newcastle.ac.uk`.

¶ Jim Webber is with ThoughtWorks, Sydney, Australia, `jim@webber.name`.

1. Introduction

The real-estate industry is slowly but surely moving towards Internet-based solutions to support various aspects of their business. The currently pursued approaches (e.g., [16]) utilise web sites to make it easier to share and discover information about properties for sale, or mortgage rates offered. In addition, XML-based standards are emerg-

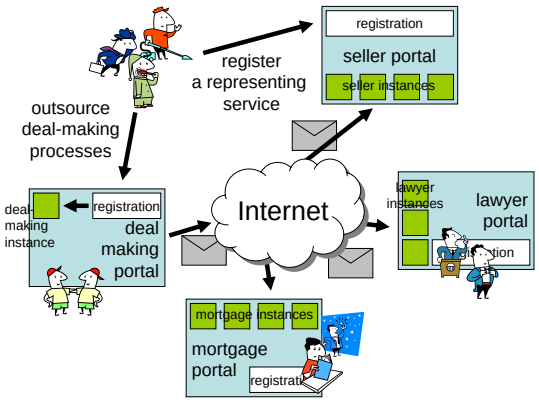


Figure 1. SOAR services landscape.

ing [9, 12, 15, 17] that support the interaction between various players (and the software packages they use), including real estate agents and mortgage lenders (see for more details Appendix A.2). Although these are good initial steps, the nature of real estate business is such that it could benefit in novel and interesting ways from more advanced service-oriented approaches to business-to-consumer and business-to-business interactions.

The objective of our work is to demonstrate how businesses and individuals can rapidly create profitable real-estate Internet services that are provably secure and correct. To that end we introduce SOAR, a Service-Oriented Architecture for the Real-estate industry. Figure 1 depicts SOAR at a high level. The main idea is that all participants in various transactions are represented by services: seller services, lawyer services, buyer services, surveyor services, etc. Services can be accessed by non-expert users through web pages for creation, configuration and termination. A service portal creates service instances when requested by users, and hosts these instances. New services can then be introduced by defining service interaction protocols in the SOAP Service Description Language and the resulting composed service can be model-checked against various liveness and deadlock properties, and has embedded a trust and security solution to assure privacy, identity and validity of user claims.

This paper describes our work during the period of the contest, from conception of the business case, to design of SOAR and the implementation of the DealMaker service. The following items are our main contributions:

- we created a business case for service-oriented computing for the real-estate industry, both for SOAR portals in general and for the DealMaker service in particular. We also argue for a possible role of standardisation bodies to successfully introduce SOAR in the diverse real-estate industry (Section 2 and Appendix A).
- we designed the SOAR architecture, with each service configurable through a web site, and personalised service instances hosted by a service provider, see Section 2.
- we suggested, designed and implemented a potential service supported by SOAR through the DealMaker service (Section 2.1).
- we embedded a security solution within SOAR to guarantee privacy, identity and validate user claims (Section 3).
- we proved correctness (with respect to the absence of starvation and race conditions) of the DealMaker service using the sequence constraints approach to protocol specification in SSDL described in Section 4.
- we implemented the DealMaker services (Section 5) and made it accessible through a demonstration web site (Appendix C and [4]).
- we designed and implemented an important new tool for the use of SSDL in managing service interaction protocols, namely an SSDL protocol execution engine (Section 5.2).

Finally, the appendices provide more details about the topics listed above, in particular with respect to the business case and the web site, and adds some reflections to our contest participation.

2. SOAR Basic Architecture

In this section we describe the main features of SOAR: basic service design, service hosting portals and personalised service instances. We also introduce the DealMaker service. First, we provide the following definitions used throughout the paper:

- *service instance* (also just *service*), see Figure 2: a run-time accessible service representation adhering to the *abstract service definition* of a particular *service type*. Service instances contain accessible *service properties*, which are stored as name-value pairs. Our security solution will provide access control at the property level.
- *service instance creation* (also just *service creation*): a *service provider* allows users to *create* (and subsequently parameterise and terminate) service instances, for the service types the provider supports. (One can think of this kind of service instance creation as the service equivalent of 'myYahoo' etc.)
- *participant*: any party involved in the system, such as lawyers, surveyors, buyers and sellers, etc., as well as the logical service representation of these parties

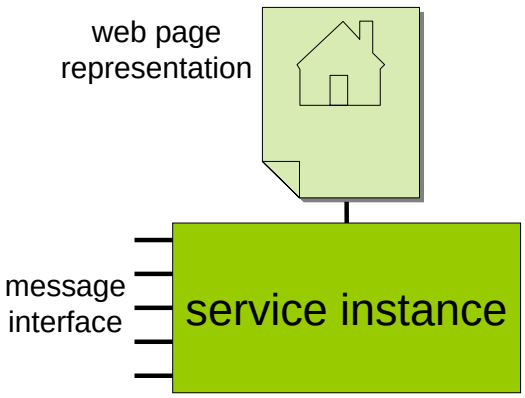


Figure 2. Service: message interfaces and web page representation.

within the system. In addition to participants, *activities* can also be represented by a service instance–example activities are drafting a contract or setting up a meeting.

In SOAR, every participant is represented by a service instance. This service contains data about the participant, and presents a messaging interface definition. The message interface allows the data to be accessed, but also allows more advanced interactions, such as ordering or stepping through stages of a workflow. The specifics of the interface definition are different for each service, and adhere to the abstract services definition for the particular participant type. Newly introduced composed services follow the same architectural design as the core services representing participants (depicted in Figure 2). That is, personalised service instances can be created and there is web page based user access to the service instances. To avoid inputting large amounts of redundant data, each service can decide to accept parameterisation referring to other service instances. For instance, a DealMaker service can be used by a buyer to create an instance that is parameterised with the service instances representing the buyer's lawyer, etc.

Every service instance contains a web page representing the service instance, and a set of message interfaces specified in SSDL (in the implementation this is translated into WSDL documents, see Section 5). The service instance executes within a run-time environment–by default, we assume that the service instances are hosted by the service provider. We imagine service providers for sellers, buyers, lawyers, etc., or combinations thereof, see Figure 1. Alternatively, participants host their own service instances, which adhere to the message interface definition for the particular abstract service.

Figure 1 gives an idea about the landscape of real-estate services we envision. We envision portals to emerge for various participant types, for instance for lawyers, mortgage companies, buyers, sellers, etc. It is very well possible that one portal supports more than one participant type. For instance, one can imagine a portal where sellers as well as buyers register. As we discuss from the business angle in Appendix A, the portal plays a key role in bootstrapping the SOAR landscape. Participants register with their respective portals, and the portals create service instances for the registrants. In Figure 1, we therefore include the box registration, which not only indicates an opportunity to register, but also implies the ensuing process of service instance creation. The portal also provides the run-time environment to host the service instances, as indicated by the instance boxes at the various portals.

There is a number of services one can think of that exploits SOAR. In Appendix A.1 we discuss them in increasing order of complexity. There we also discuss the business case behind such services as well as behind SOAR itself.

2.1. The DealMaker Service

The DealMaker service is a complex service that demonstrates the abilities of SSDL, its associated formal proof system, and our security model. The DealMaker service helps customers to go through the steps involved in buying and selling real-estate. We have taken the process example from [7]. The service can be instantiated by any party, but for the sake of this explanation, we assume the buyer initiated the creation of a DealMaker service instance. At initialisation, it will be parameterised with the necessary information about parties involved in the deal making, such as lawyers, mortgage providers, surveyors, etc. Then, it goes through the process steps. To get an idea about the operation of the DealMaker service, it is probably simplest to read the SSDL specification of the DealMaker service given in Figure 3. The main protocol, named `Buy_Sell_Protocol`, contains a sequence of steps, each referring to another protocol: organising the mortgage, organise property viewing, add lawyer information, price negotiation protocol, etc. The stages corresponding to valuation and surveying can be executed in parallel, as one can see in Figure 3. At the end of the process, the contract gets exchanged.

We note that the DealMaker service does not attempt to *completely* automate stages of a business process. On the contrary, the assumption is that the human stays involved at all time, and many of the individual protocol steps given in Figure 3 contain status update messages sent to the right parties at the right time to assure completion of the overall process. The human then has to act on these messages for the `Buy_Sell_Protocol` to continue, and ultimately complete. In Figure 4 we display the details of the mortgage organisation protocol, as SSDL specification. It has two participants involved, the buyer and the mortgage lender. When

```
<?xml version="1.0" encoding="UTF-8"?>
<ssdl:protocol targetNamespace="http://www.ncl.ac.uk/DealMakingService/ContractExchange/protocol"
     xmlns:msgs="http://www.ncl.ac.uk/DealMakingService/ContractExchange/messages" xmlns:sc="
    urn:ssdl:protocol:sc" xmlns:ssdl="urn:ssdl:v1">
  <sc:sc>
    <!--Message Exchange Protocol-->
    <sc:protocol name="Buy_Sell_Protocol">
      <sc:sequence>
        <sc:protocolref ref="MortgageOrganiseProtocol"></sc:protocolref>
        <sc:protocolref ref="ViewingOrganizeProtocol"></sc:protocolref>
        <sc:protocolref ref="LawyerRegisterProtocol"></sc:protocolref>
        <sc:protocolref ref="SearchProtocol"></sc:protocolref>
        <sc:protocolref ref="PriceNegotiationProtocol"></sc:protocolref>
        <sc:parallel>
          <sc:protocolref ref="ValuationProtocol"></sc:protocolref>
          <sc:protocolref ref="SurveyProtocol"></sc:protocolref>
        </sc:parallel>
        <sc:protocolref ref="LifeAssuranceProtocol"></sc:protocolref>
        <sc:protocolref ref="MortgageConfirmationProtocol"></sc:protocolref>
        <sc:protocolref ref="ContractExchangeProtocol"></sc:protocolref>
      </sc:sequence>
    </sc:protocol>
  </sc:sc>
</ssdl:protocol>
```

Figure 3. SSDL specification of the protocol followed by the DealMaker service.

the seller initiates the creation of a DealMaker service instance, it parameterises the service instance by providing buyer and lawyer information. Importantly, it does not just provide a name, but a reference to the service representing the buyer and lawyer.

The service provider that hosts DealMaker services manages the interaction given in the SSDL specification of the DealMaker service. To that end, an SSDL protocol engine runs at the service provider. It tracks how far the process is along, and initiated next steps as appropriate. The SSDL protocol execution engine is further discussed in Section 5.2.

3. Trust and Security Architecture

The SOAR architecture requires solutions for common security issues such as authentication, privacy, etc., which we discuss this in Section 3.1. However, of more specific interest to SOAR is the issue of achieving trust about claims of unknown participants in a transaction, such as about home ownership or professional credentials. We designed a SAML-based trust solution for participant claims, which we discuss in Section 3.2.

3.1. Authorization, Confidentiality and Integrity

The communication among services and between services and web users is done using SSL [6], providing basic security properties such as confidentiality and integrity. The assumption is that all service providers have acquired X.509 certificates [8], issued by a valid CA. However, SSL alone is not sufficient for identification, authentication and authorization within services instances. Therefore, our security model uses SAML assertions [10] to provide identity as well as authenticity in message exchanges. The authorization is done by role-based access control [5] mechanism, where "roles" and "rights" are provided through SAML attribute assertions. With SAML we establish a standardized way to share credentials and an easy way to include new services or users into the system.

Service instances may have various properties that need to be protected. For instance, a seller may only be willing to share information about his/her lawyer with participants that are trying to close a deal, i.e., with services that are in same DealMaker service instance. Hence, when a new DealMaker service instance is created, each participant of this instance will receive a SAML attribute assertion (a role, e.g., `dmi:ID648s5e2:participant`), indicating that they are allowed to access "protected properties". The default access control policy defines restrictions to some service properties, and this policy is then updated to reflect new service instances. For illustration, Figure 5 presents a small piece of our access control policy.

We also want to be able to hide the identity of the 'real person' that is behind a SOAR participant. In our model, the real identity of a person will be known only by the particular portal the service is created with. To other participants, a person's identity will always be obfuscated by referring to the person through a service identifier.

3.2. Trusted Claims

In SOAR, individual participants could make unsubstantiated claims about ownership of properties, etc. In real life

```
<?xml version="1.0" encoding="UTF-8"?>
<ssdl:protocol targetNamespace="http://www.ncl.ac.uk/DealMakingService/MortgageOrganise/protocol"
     xmlns:msgs="http://www.ncl.ac.uk/DealMakingService/MortgageOrganise/messages" xmlns:sc="
    urn:ssdl:protocol:sc">
  <sc:sc>
    <!--Parties In Mortgage Organise Protocol-->
    <sc:participant name="Buyer" />
    <sc:participant name="MortgageLender" />
    <!--Message Exchange Protocol-->
    <sc:protocol name="MortgageOrganiseProtocol">
      <sc:sequence>
        <sc:choice>
          <sc:sequence>
            <ssdl:msgref ref="msgs:MortgageRequestSubmission" direction="in" sc:participant="
                Buyer" />
            <ssdl:msgref ref="msgs:MortgageRequestTemplate" direction="out" sc:participant="
                MortgageLender" />
            <ssdl:msgref ref="msgs:MortgageRequestCompletedTemplate" direction="in"
                sc:participant="Buyer" />
            <sc:choice>
              <sc:sequence>
                <ssdl:msgref ref="msgs:MortgageRequestAccepted" direction="out" sc:participant=
                    "MortgageLender" />
              </sc:sequence>
              <sc:sequence>
                <ssdl:msgref ref="msgs:MortgageRequestRejected" direction="out" sc:participant=
                    "MortgageLender" />
              </sc:sequence>
            </sc:choice>
          </sc:sequence>
          <sc:nothing />
        </sc:choice>
      </sc:sequence>
    </sc:protocol>
  </sc:sc>
</ssdl:protocol>
```

Figure 4. SSDL specification of the protocol followed in the mortgage organisation step.

```
<policy>
  <resource id="lawyer" defaultAction="deny">
    <allow>
      <role id="dmi:ID648s5e2:participant" />
      <role id="dmi:ID24n256s:participant" />
    </allow>
  </resource>
</policy>
```

Figure 5. Access control policy.

we can often easily enhance trust in such claims (such as ownership of a house) by paying a personal visit or search government archives to check if the supplied claim is true or not. However, in the virtual world of SOAR, services are often not in a position to make judgement calls about a claim of a participant is valid, possibly simply because no humans are available with the right expertise. To protect SOAR from illegitimate entrees, we use a Trusted Third Party (TTP) that is able to corroborate the claim of a participant. We can think for instance of a government institution being able to issue *claim tokens* that substantiate the claims about the ownership of a real-estate property made by a particular seller.

For example, before the creation of a service instance that offers a house for sale to all SOAR participants, the seller needs to supply house details and a claim token issued by some claim-issuing institution, indicating that the house details can be trusted. Let us assume that the seller goes in person to a government institution to show a “legal document” indicating ownership of his/her house. The government then gives the seller a claim token that the seller can forward to other SOAR participants, who then can check the validity of the claim token at the claim issuer web service.

In the demo we apply the idea of claim tokens to properties associated with a potential buyer that chooses to make use of the DealMaker service. Our implementation, based

on SAML assertions, provides a flexible and user-friendly way for participants to either obtain or check claim tokens. We think that trusted claims provide a level of trust throughout the SOAR architecture that may greatly enhance the willingness of participants to carry out business interactions through SOAR services.

4. SSDL and Formal Correctness Proof

The DealMaker service constitutes of a particularly complex orchestration of service interactions. The complex nature of the interactions makes one question the correctness of the overall process. In order to validate the correctness, we derive a π–calculus specification from the SSDL specification, and validate the resulting model formally. The way this can be done has been described in [13], and we briefly summarise the main points of this approach to correctness validation. First we introduce SSDL, closely following [13, 18].

The SOAP Service Description Language (SSDL) is a SOAP-centric contract description language for Web Services. The SOAP Service Description Language provides the base framework for a range of protocol description frameworks which at one end of the spectrum can be a simpler, SOAP-focussed, direct replacement for WSDL message exchange patterns while at the other end of the spectrum can enable formal validation and reasoning about the protocols that a Web Service supports. SOAP is the standard message transfer protocol for Web Services. However, the default description language for Web Services (WSDL) does not explicitly target SOAP but, instead, provides a generic framework for the description of network-exposed software artefacts. Another important feature of SSDL is the ability to specify multi-party protocols that are considerably more complex than the simple message exchange patterns allowed in WSDL. In SOAR we utilise the sequencing constraint manner of specifying protocols, which makes the ensuing protocol amenable to formal correctness verification.

Figure 3 and Figure 4 illustrate the use of the sequencing constraint protocol definition (the sequencing constraint schema is specified in the namespace ending with `sc`). The use of sequencing constraints results in a protocol that can be formally expressed in terms of π–calculus, thus allowing for model-checking tools to demonstrate correctness. The formal correctness proof considers the following properties: race conditions and starvation. One can also consider if an agreed-upon termination state will be reached, but we did not pursue this in this project. A race condition emerges if different participants observe different paths forward, for instance when a sender knows a message has been sent out, while the receiver assumes no message has been sent out since it has not arrived yet. In this case, sender and receiver might take different next steps in the protocol. Starvation occurs when contracts are incompatible because certain message assumed by a receiver are not part of the protocol of the assumed sender.

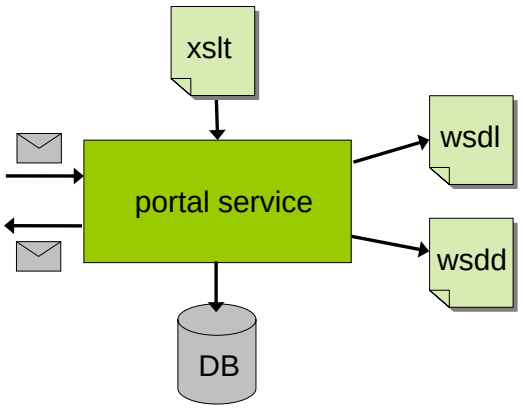


Figure 6. Service instantiation process.

We used SSDL to validate the lack of race conditions in an early version of the DealMaker service protocol specification given in Figure 3. Further details are provided in Appendix B.4.

5. Implementation and Run-Time Environment

In this section we discuss two major elements of our implementation, the service run-time environment in Section 5.1 and the SSDL protocol execution engine in Section 5.2. Extended versions of both sections can be found in Appendix B.

5.1. Service Run-Time Environment

We subsequently discuss instantiation, deployment and invocation of services.

5.1.1. Service Instantiation The concept of a portal in our architecture facilitates participants in the real-estate industry to create service instance representing them and their constituent properties. The functionality behind each of the service instances is analogous, and the sole distinguishing factor in each is the data “contained” within in. To provide a replicated service implementation for each service instance would be inefficient. We pursued a more elegant and efficient solution to this issue by providing a specialised interface to a generic service, enabling reuse of the service implementation, yet retaining the notion of distinct service instances.

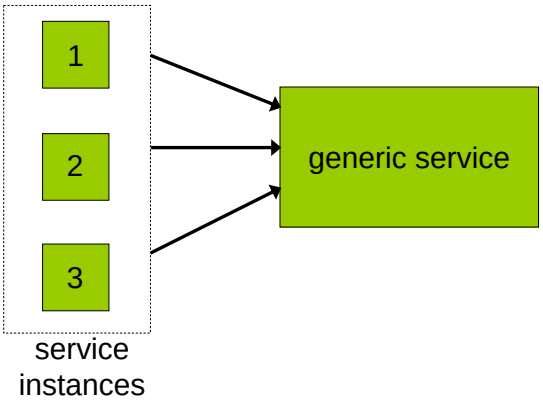


Figure 7. Service deployment process.

The production of the specialised interface, and thus service instantiation is performed by an operation at the portal service. This operation receives the instance-specific data in XML format, within a SOAP envelope, from the invoking party, be it another service or a front-end to the portal service. We use XSLT [19] to process the data received since it offers a highly effective means of focused data extraction and template incorporation. In the production of the specialised interface we wish to create for each service instance, we simply plug the instance-specific data into spaces left within a WSDL template. Within the WSDL template, the transformation simply customizes the name of the service, and endpoint at which this service was deployed.

Buyers and sellers (and other participants) can state, when registering at the appropriate portal, the service representing their lawyer, surveyor etc, for use in the deal-making process. This statement is made in the form of the URL to the given service WSDL, resulting in a list of WSDL URLs behind each buyer and seller service instance. In collecting a number of services together and making them available through a single interface, we have created a very straightforward form of service composition. Figure 6 depicts the various aspects of service instantiation.

5.1.2. Service Deployment The final output of the above instantiation process is the WSDD document. It is within this document that we compose our service instance, stating the generated WSDL as the service interface and the generic service as the implementation. Figure 7 shows how each of the created service instances is linked to the generic service implementation. With these details we have a complete description of the service instance, and therefore this service may be deployed. A tool within Axis is then used to deploy the service and enable its invocation by relevant parties. Figure 8 shows an example deployment file for the buyer and seller services.

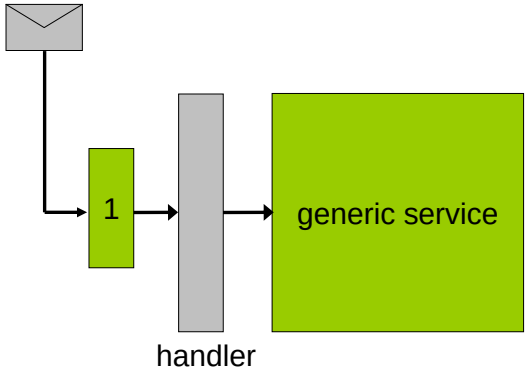


Figure 9. Service invocation process.

5.1.3. Service Invocation Deployment in above described way requires that the appropriate context be forwarded to the generic service, to enable it to distinguish invocations for different service instances. That is, given invocation of service instance A, we convey to the generic service implementation I, that context should relate to A. Therefore, all messages sent to the endpoint of a given service instance, must first pass through a handler, before being forwarded on to the generic service implementation (see Figure 9). The handler, on receipt of a message directed at a service instance endpoint inspects the message destination, that is, the endpoint of the service instance. With the use of a unique identifier for each service instance (incorporated into the instance endpoint URL), the context for a message can be derived from the message destination. This context is then added into the message body, by the handler, providing the necessary context to the generic service implementation. With this context in place, the message is safely forwarded on to the service implementation for processing. This approach shows how context for service invocation can be made implicit from the service instance endpoint rather than being included explicitly within the message. This enables the creation of replicated service instances, linked to the same service implementation, which behave as if a stand alone service with specific implementations.

5.2. SSDL Protocol Execution Engine

SSDL fully describes the state space of a composed service as well as the sequence of service interactions (message exchanges) required to reach each state. We can thus view a composed service whose description is given in SSDL as a state machine, with message exchanges providing the transitions between states, and states implicitly defined as points between these exchanges. Starting from this premise,

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/" xmlns:java="http://xml.apache.org/axis/wsdd/
    providers/java">
  <service name="Package_n" provider="java:RPC" style="rpc" use="encoded">
    <parameter name="className" value="scc2006.packages.PackageService"/>
    <wsdlFile>wsdl/Package_n.wsdl</wsdlFile>
    <parameter name="allowedMethods" value="*"/>
    <requestFlow>
      <handlertype="java:scc2006.packages.PackageHandler"/>
    </requestFlow>
  </service>
</deployment>
```

Figure 8. Sample service instance deployment file.

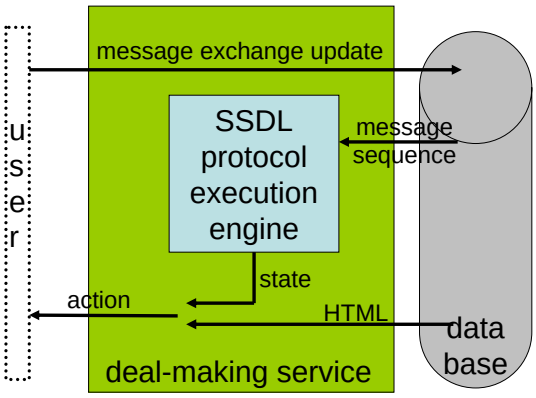


Figure 10. The SSDL Protocol Execution Engine within the DealMaker service.

we developed an SSDL Protocol Execution Engine that directly executes the state machine defined by the SSDL description.

SSDL documents describe the state space in the form of a tree whose leaf nodes are <msgref> elements. These specify that the type of message referenced in their ref attribute be sent or received. Other elements (sequence, parallel, branch, loop) define the order in which message exchanges in their subtrees need to occur. The protocol engine must correctly implement the semantic of the different elements. How this is done is discussed in detail in Appendix B.2.

The general architecture of the engine is shown in Figure 10. Based on the state reported by the SSDL Process Execution Engine, the DealMaker invokes actions tied to each state. In addition, it offers facilities to keep the internal machine state consistent with the deal's real-world status. If a user request so, based on the machine state, the DealMaker service retrieves an explanatory, pre-generated HTML page from the database and delivers it to the user.

5.2.1. State-keeping in a stateless environment Web Services that use Axis RPC wrappers are inherently stateless. Every service invocation starts with a freshly-loaded executable. One way to keep state is by storing the input sequence that was encountered previous to reaching the current state. To restore state, the engine then steps through this sequence, ignoring actions tied to the states it traverses.

In regard to reaching the current state after startup, this method is clearly less efficient than explicit state-keeping, because all steps of the machine have to be executed again before the actual action invoked can be taken. However, we used it because (a) it is more flexible, and (b) helps to implement fault-tolerant applications (see Appendix B.2). Higher flexibility results from the fact that the state machine description (the SSDL document) can be modified between service invocations, without necessarily invalidating any partially-completed processes. This is of particular importance with long-term processes such as that implemented by the DealMaker, where one process instance may be running for several months before all steps have been completed.

6. Conclusion

This paper reports on two and a half months of team work on service-oriented computing, which included conceiving the idea of the DealMaker service, researching the real-estate business domain, designing the SOAR architecture including extensive security and trust solutions, implementing the DealMaker service and the supporting SSDL protocol execution engine, and applying model checking to a version of our protocol. The work combines state-of-the-art fundamental computer science approaches with practical implementation and with the business and standardisation side of such work. It leverages deep skills of the team members in security, protocol specification, formal methods and service-oriented software engineering, as well as the business experience of the senior team members. We believe that the current work provides a useful exploration in applying service-oriented computing technologies in the real-estate industry, with some exciting ideas and challenges we hope to continue working on in the future.

Acknowledgement

Andrew Fletcher started the contest work with us, but unfortunate circumstances left him no choice but to pull out. We have very much appreciated his enthusiastic interest in the project as well as the input he was able to give in the early design stages.

References

[1] Apache. Apache XML Security. `http://xml.apache.org/security`.

[2] Apache. WSS4J XML Encryption and XMLDigitalSignature. `http://ws.apache.org/wss4j`.

[3] Apache. *Axis Architecture Guide v1.2*. The Apache Software Foundation, 2005.

[4] E. de Mello, S. Parastatidis, P. Reinecke, C. Smith, A. van Moorsel, and J. Webber. Demo of SOAR: Close the Deal. `http://vs-soc.ncl.ac.uk:8180/demo/index.html`, 2006.

[5] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security*, 4(3):224–274, 2001.

[6] A. O. Freier, P. Karlton, and P. C. Kocher. *The SSL protocol - v.3*. Internet Draft, Março 1996.

[7] home.co.uk. Home Buying Guide: Introduction. `http://www.home.co.uk/guides/buying`.

[8] R. Housley, W. Polk, W. Ford, and D. Solo. *Internet X. 509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. IETF RFC 3280, Abril 2002.

[9] MISMO. Mortgage Industry Standards Maintenance Organization. `http://www.mismo.org`.

[10] OASIS. *Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) v1.1*. Organization for the Advancement of Structured Information Standards (OASIS), Setembro 2003.

[11] OpenSAML. `http://www.opensaml.org`.

[12] OSCRE. Open Standards Consortium for Real Estate. `http://www.oscre.org`.

[13] S. Parastatidis, S. Woodman, J. Webber, D. Kuo, and P. Greenfield. Asynchronous Messaging between Web Services Using SSDL. *IEEE Internet Computing*, 10(1):26–39, Jan/Feb 2006.

[14] PISCES. Interoperability: The Movie. `http://.pisces.co.uk` and `http://www.oscres.org`.

[15] PISCES. Property Information System Common Exchange Standard. `http://www.pisces.co.uk`.

[16] Real Estate Web Sites. `http://www.imoscout.de`, `http://www.immonet.de`, `http://www.immowelt.de`.

[17] RETS. Real Estate Transaction Standard. `http://www.rets.org`.

[18] ssdl.org. SSDL–The SOAP Service Description Language. `http://www.ssdl.org`, 2005.

[19] W3C. XSL Transformations (XSLT). `http://www.w3.org/TR/xslt`, 1999.

(Appendices that complete the report start on next page.)

Appendices

The appendices to the above paper complete the overall contest report. In these appendices we provide more details about the following topics:

- Appendix A: the real estate problem domain and value proposition behind our SOAR and SOAR services
- Appendix B: report on implementation experiences and lessons learned, including the service run time in Appendix B.1, the SSDL execution engine in Appendix B.2, the security solution implementation in Appendix B.3, and the formal validation of SSDL in Appendix B.4
- Appendix C: explains the idea behind the demo web site in [4]
- Appendix D: a result analysis and reflection on the project

A. Problem Domain and Value Proposition

We discuss in Appendix A.1 the business case behind SOAR and individual services such as the DealMaker services, and we review in Appendix A.2 the state-of-the-art of computing technologies used in the real-estate industry.

A.1. Value Proposition

In this section we discuss the business case behind SOAR as well as individual services. It will turn out that the dynamic nature of the real-estate industry may make introduction of SOAR technologies through a winner-take-all portal relatively difficult. As a consequence, we argue that there is a role for standardisation bodies within the real-estate industry to expand their work into defining industry-wide service-oriented message interfaces.

The real-estate industry is a particularly diverse and dynamic industry. As an industry, it is not only concerned with sales and purchases of individual properties, but also of industrial properties and in response to requests for proposals in larger real-estate deals. Many parties are involved, individuals as well as businesses, with buyers and sellers continuously changing, and with a high amount of novices participating. Many steps in a real-estate transaction are traditionally human-intensive, such as viewing, decision-making, negotiating, etc. There is a continuous concern about trust and security: trust in other parties, security concerns about identity and privacy. Moreover, there is a concern of trust in automation as well as trepidation of new buyers and sellers to step into the unknown world of real-estate. Regional knowledge is important to be a successful real-estate agent, and laws and regulations are different in every country or state, and are subject to change from time to time. Because of this diversity and dynamism in the real-estate industry, it is perhaps not surprising that automation and Internet-based cooperation are relatively slow to emerge. Compared with supply-chains or resource planning, the domain is much less straightforward to automate. However, it is exactly for these reasons that service-oriented computing solutions are required to answer the domain challenges of the real-estate business.

We believe that service-oriented computing has the potential to disrupt the real-estate industry by enabling new business practices that alter the role of current players. As an example, the role of a real-estate agent might change because of match-making and information-sharing capabilities of Internet technologies. This is already true with plain web sites, but becomes even more apparent if service-oriented solutions arise. Instead of considering this disruption a threat, we look at this as an opportunity. By adding service-oriented capabilities, new businesses can be conceived that increase the effectiveness and abilities of an existing real-estate agent. For instance, agents might rapidly create services for specific geographic areas–these service include, but are not limited to, the brochure service and listing services mentioned in the introduction.

Potential Real-Estate Services in SOAR We illustrate the potential of SOAR by describing some example applications, one of which we implemented (the 'DealMaker service'). As a first example, assume the service to be hosted on the Internet, and assume that there is a SOAR portal that hosts information from sellers, including information about their properties, and from real-estate agents (including information about their company). Using the message interfaces of the services, one can then relatively easily create new services.

For instance, one can create a 'brochure service', which at the request of a buyer or agent selects a set of houses, prints them out including personalised logo and other information from a real-estate agent, and mails them to selected customers of the real-estate agent using e-mail as well as regular mail. Clearly, if such a service would have to be built using information from web sites without agreed-upon messaging interfaces, it is very difficult to build at best. There is potential for many other information-centric services that utilise the message interfaces of the portals, such as listings dependent on geographic areas, generation of targeted advertisements, etc.

Things get even more interesting when the messages not only access information, but initiate actions, such as drafting a contract or setting up a house viewing. This is illus-

trated by the DealMaker service in Figure 1. It provides the same service instance creation possibilities as the other services, but when executing, the service utilises other services to complete its process of closing a deal. Messages may for instance start the process of drafting a contract. All together, the DealMaker service does as many automatable process steps as possible to close a real-estate deal: arranging viewing dates, contacting surveyors, exchanging mortgage information, providing information for the contract, etc. The DealMaker service is described at length in Section 2.1.

Business Case for SOAR. With respect to the SOAR service architecture we introduce in this paper, there are different perspectives that one can take in judging the business validity of the approach. First, one can consider SOAR in business-to-business and even Intranet setting. In that case, one does not need to consider the difficult to control dynamics that individual customers introduce to SOAR. This greatly simplifies the bootstrapping challenge of SOAR (see below in this section for a detailed discussion of the bootstrapping issue). After all, large companies or collaborating companies can simply choose to use SOAR, because of increased efficiency and flexibility over web sites, XML interchange formats and also object-oriented methods. The trade-offs in choosing a particular architecture are similar to any other industry: ease of use, legacy issues, etc. To truly advance the state of the art, standardisation of message interfaces is necessary, either in de facto manner or through standardisation bodies such as OSCRE and PISCES [12, 15].

Although there certainly is ample reason to introduce SOAR-style architectures in business-to-business or Intranet settings, we designed SOAR with the open Internet in mind. Customers can be private individuals or businesses, thus leading to business-to-consumer as well as business-to-business scenarios. Newly provided services would be expected to aim at attracting private customer as well as small and medium businesses like lawyers, real-estate agents, surveyors, etc. All these participants will be represented by a service, which they personalise themselves through a web site. In this set-up there are two major business issues: how can SOAR be bootstrapped, and how can one make money out of starting individual added-value services such as the DealMaker service? We discuss both.

Bootstrapping SOAR. The Achilles heal of SOAR is gaining initial sustainable acceptance of the portals and the service interface definitions. We refer to this as the bootstrapping problem. The dilemma is that to start a successful portal one needs customer, and to attract customers one needs a successful portal. One obvious solution to this is to advocate the emergence of a dominating, winner-take-all, service portal–this could be a portal for each type of participant (lawyer, seller, buyer, etc.), or a portal that covers all participants. Once the portal emerges, it can introduce de facto standards for interoperability along the lines of SOAR. This model mimics e-Bay, amazon and 'vertical portals' such as in the car or high-tech industry. At first sight, one might think it is a matter of time until a major real-estate portal will assume such a winner-take-all position. However, we doubt if a winner-take-all solution is likely to materialise in the real-estate industry. The real-estate industry is much more diverse than the publishing or music industry, and local cultural as well as legal aspects are different in every geographic zone. It then becomes very challenging to create a portal that is attractive for a large customer base.

If a dominant portal does not emerge, an important role arise for the standardisation bodies that represent the real-estate industry [12, 15]. Without a dominant players, de factor standardisation of interfaces is not likely to happen, and a standardisation organisation can fill the void. The organisation can also venture in creating initial portals, this resolving the SOAR bootstrapping issue. Commercial, local portals can then be expected to emerge, and creation of new value-added services would start and significantly improve the position of the real-estate industry.

Business Case for the DealMaker Service The DealMaker service is just one example of a service that can be created within the SOAR landscape. The business case behind these added services can be manifold: it can be based on charging for transactions, it can use subscription-based charging of its customers, or it could target advertisements. And, of course, a combination of these is possible. For added-value services to emerge, portals must allow messages to be sent to their hosted services, and the interfaces must be stable, preferably standardised. Here again one sees the value of standardisation, either de jure through a standardisation body, or de facto through winner-take-all portals.

A.2. Real Estate and Internet: State of the Art

As many other industries, the real estate industry is pursuing standardisation efforts to facilitate interoperability among various participants. Arguably, the real-estate industry is somewhat behind various other industries in such areas as high tech supply chains, automobile portals, etc. (the real-estate industry itself admits so in [14]). It is not unlikely that the challenges we recognised in terms of making interoperability work have something to do with the reason standards are relatively slow to take off: many different player, many different role, many different regulations, different in every geographical region. However, some serious interoperability efforts have been started, of which we mention a few.

Other obvious utilisation of the Internet is through web sites that provide information about real estate properties, allows people to advertise their property, etc. Such web sites are emerging in many countries (for some example in Germany, see [16]), states and cities, again demonstrating the dependence on geographics. These web site are a good start, but tailor to mass markets and do not provide additional interaction capabilities for software packages.

Directly related to the real-estate industry are the not-for-profit organisation Property Information System Common Exchange Standard (PISCES [15]) in Europe and the Open Standards Consortium for Real Estate (OSCRE [12]) in the United States. These organisations are publishing XML standards for interoperability between real-estate agents and possibly other parties. The agreed-upon XML formats for describing real estate related information can directly be plugged into our SOAR architecture to hook our work into existing developments. However, both PISCES and OSCRE are less concerned with the software architecture needed to deliver on some of the automation promises mentioned in the vision the expose to the world [14]. Discussions are underway what type of transport to use (e.g., ebXML or other web service technologies), but that stops far short from a service-oriented solution.

Many other standardisation efforts are of interest, to name a few, the Mortgage Industry Standards Maintenance Organization (MISMO [9]) for the mortgage industry or the Real Estate Transaction Standard (RETS [17]. The latter is centred around a server platform solution, but in general these organisations propose standardised formatting of data in XML, so that software programs can easily interoperate. We have argued in this paper that new opportunities arise if one goes beyond format and looks towards flexible ways of enriching the interaction between participants. Nevertheless, a very important role could be in store for the standardisation bodies to push the XML standardisation of message-based interactions. The creation of XML standards for the message interchanges by standards bodies would be an alternative to the emergence of large portals that create their own message interchange definitions which will turn into de facto standards. Because of the diversity for the industry, large commercial players may be slow to emerge, in which case there is a role for standardisation bodies to push SOAR like solutions for the real-estate industry.

B. Implementation Experiences and Lessons Learned

We implemented our solution using Java technologies, a choice motivated by the familiarity with the technologies and the existence of diverse tool support. The demo runs on a web server hosted at The University of Newcastle, relying on Axis and on a mySQL database. Axis and related software runs in VMWare virtual server on Linux (offering a Linux guest OS for our use), connected to the database hosted on another machine meant for student projects. Note that our service implementation is not distributed over many hosts but are web service all on the same host (identified by URLs). The choice of individual development environments were left to each developer in the team. We made limited use of UML techniques to describe and share designs, without any particular tool support in that area. The web site was implemented by relatively standard means: dynamic web pages, using 'AJAX'-style interaction through Javascript and passing of XML documents.

B.1. Service Run-Time Environment

This subsection is an extended version of Section 5.1.

The SOAR implementation necessitates a run-time environment capable of supporting the characteristics synonymous with the service-oriented paradigm. The architecture used to support service-orientation characteristics is web services, and principally the SOAP-based implementation of this architecture. Numerous tools supporting the development of web services have been born out of this position at the forefront of service-orientation, and the SOAR implementation takes advantage of one such tool, namely Apache Axis [AXIS]. Axis provides a sound basis for web service development with, most notably, the provision of a SOAP processing and transport framework and flexible service deployment options. This utilisation of Axis enabled a certain degree of abstraction to be achieved in the development process, with focus shifting, as far as possible, to higher level and more conceptual notions.

Standard techniques for web service instantiation, deployment and invocation are ubiquitous, as the adoption of the service-oriented paradigm and web services architecture gains ever-increasing momentum. Consequently, this section places focus on the innovative, non-standard techniques used in our implementation approach in relation to service instantiation, deployment and invocation.

B.1.1. Service Instantiation One fundamental requisite of a web service is the decoupling of interface from implementation, facilitating opaque invocation of web services and a focus on what functionality is provided not how it is provided. The interface states the operations and message formats supported by the service, and the endpoint at which this service resides. The service consumer is abstracted away from service implementation details, and concerns himself with only matters prior to message dispatch to the endpoint. Such abstraction provides a high degree of flexibility to the service provider, allowing service implementation to be arbitrarily complex whilst maintaining a

consistent and abstract interface. In the SOAR implementation we capitalise on the flexibility offered by this decoupling, and exploit the power of this technique for service instance creation at run-time.

The concept of a portal in our architecture facilitates participants in the real-estate industry to create service instance representing them and their constituent properties. One can view this creation as a factory-style process, a participant registers at the portal, and portal creates a service instance for them representing their particular participant type, for instance the buyer portal would create a service instance for a buyer. The functionality behind each of these service instances is analogous, and the sole distinguishing factor in each is the data contained within in. To provide a replicated service implementation for each service instance would not only be highly inefficient, it would also be contradictory to the core principles of service-orientation; replicating rather than reusing functionality. A more elegant and efficient solution to this issue would be to provide a specialised interface to a generic service, enabling reuse of the service implementation, yet retaining the notion of distinct service instances.

The production of the specialised interface, and thus service instantiation is performed by an operation at the portal service. This operation receives the instance-specific data in XML format, within a SOAP envelope, from the invoking party, be it another service or a front-end to the portal service. Contrary to all other service operations within the SOAR implementation, this operation utilised document-style, literally encoded SOAP messages. The justification for this stems from our wish to use XSLT [19] to effectively and concisely process the data received. RPC encoded messages would not be suitable for this purpose, as their component data is isolated on receipt and made accessible as atomic data items.

XSLT offers a highly effective means of focused data extraction and template incorporation. This makes it highly effective in the production of the specialised interface we wish to create for each service instance simply plugging the instance-specific data into spaces left within a WSDL template. The functionally analogous nature of the service instances meant that a set WSDL template contained all the pre-defined operations and message formats, and the transformation simply customized the name of the service, and endpoint at which this service was deployed. The instance-specific data "behind" the service instance must be stored in a database to enable its retrieval and amendment by subsequent instance invocations. Each service instance is assigned a unique identifier and the data is linked to this identifier within the database. We again perform the task of data extraction with the aid of XSLT, which plugs the extracted data into a pre-formed SQL statement and updates the database accordingly. Use of templates in this way offers a high degree of flexibility to change, as new templates can be plugged in as requirements evolve.

The final output of the instantiation process is the WSDD document, generated using precisely the same method as the WSDL document, through use of a template in XSLT. Axis offers flexible deployment through customizable options with this WSDD document, enabling the definition of numerous service-specific details including the interface and implementation corresponding to this service. It is within this document that we compose our service instance, stating the generated WSDL as the service interface and the generic service as the implementation. Further explanation of the deployment process is left to the next section.

Worthy of further discussion is the instantiation of buyer and seller service instances. Buyers and sellers can state, when registering at the appropriate portal, the service representing their lawyer, surveyor etc, for use in the deal-making process. This statement is made in the form of the URL to the given service WSDL, resulting in a list of WSDL URLs behind each buyer and seller service instance. These addresses are stored within the database along with any other instance-specific data and can be extracted for use in the deal-making process. In collecting a number of services together and making them available through a single interface, we have created a very straightforward form of service composition.

One may, of course, provide a specialised, custom implementation for a given service instance, as would most likely be the case for mortgage lenders, lawyers etc. For the case of buyers and sellers though, it is unlikely that the resources and knowledge available would enable them to configure a specific web service for themselves. Our approach, therefore, strives to illustrate the elegance and efficiency with which functionally analogous service instances can be created by services at run time, using pluggable templates. This holds many opportunities both within and outside of the real-estate industry.

B.1.2. Service Deployment As discussed in the previous section, one output of the instantiation process is a WSDD document, enabling the custom deployment of created service instances. The document generated is used by Axis to correctly deploy the service, and to route service invocations to the appropriate service implementation.

Figure 7 shows how each of the created service instances is linked to the generic service implementation. We establish this configuration in the WSDD document, stating the generic service implementation as the implementation of the service instances, and the generated WSDL as the interface for this service. With these details we have a complete description of the service instance, and therefore this service may be deployed. A tool within Axis is then used to deploy the service and enable its invocation by relevant parties. Figure 8 shows an example deployment file for the

buyer and seller services. Deployment in this way requires that the appropriate context be forwarded to the generic service, to enable it to distinguish invocations for different service instances. We discuss this notion of context with regard to service invocations in the next section.

B.1.3. Service Invocation This invocation model of web services can be extended with the notion of handlers. Handlers enable web services to define a functional intermediary in the invocation process to intercept all incoming and/or outgoing messages, execute some functionality, and on completion forward the message on. This functionality is commonly used to enforce security or trust procedures before the invocation of a web service, but within the service instance creation process we use these handlers to convey context.

Our use of a generic service implementation for multiple service instances requires that context be conveyed. That is, given invocation of service instance A, we must convey to the generic service implementation I, that context should relate to A. Of course, this could simply be included in the SOAP communication to the service instance, but this is contradictory to the idea of generating a specific service instance. In such a case, we could simply have one generic service interface and implementation, and pass instance context as a parameter to this service, in the form of the instance identifier. The SOAR implementation endeavoured to create a more elegant ad useful approach to this context communication, and found such an approach in the use of handlers.

Our context for each instance was the instance identifier, and it was this identifier we required to be conveyed to the generic implementation. Messages arriving at the service instance endpoint had no containing context, that is, the context was implicit from the endpoint at which the message was directed. For instance if we dispatch a message to the endpoint of service instance A, we do not include within the body of that message any reference to service instance A. If such a message was then simply forwarded on, without amendment, to the generic implementation, we would be unable to derive the message context, that is, the instance to which this message relates.

To deal with this notion of context we introduced a handler to the invocation process. All messages sent to the endpoint of a given service instance, must first pass through this handler, before being forwarded on to the generic service implementation. The handler itself is generic, and the same handler is utilised by all service instances, and in essence this handler can be seen as part of the generic implementation. The handler, on receipt of a message directed at a service instance endpoint inspects the message destination, that is, the endpoint of the service instance. With the use of a unique identifier for each service instance (incorporated into the instance endpoint URL), the context for a message can be derived from the message destination. This context is then added into the message body, by the handler, providing the necessary context to the generic service implementation. With this context in place, the message is safely forwarded on to the service implementation for processing. Such an approach has shown how context for service invocation can be made implicit from the service instance endpoint rather than being included explicitly within the message. This enables the creation of replicated service instances, linked to the same service implementation, which behave as if a stand alone service with specific implementations.

B.2. SSDL Protocol Execution Engine

This section is an extended version of Section 5.2.

There already exist various tools to support the use of SSDL in the development of new web services. These tools provide facilities for correctness-checking SSDL descriptions and for the automated creation of .Net stubs from SSDL. As with most similar approaches throughout the field of computer science, however, a gap opens between formally checked descriptions and their actual implementation, because the components involved must still be developed manually Hence, mistakes during the implementation could re-introduce protocol errors easily found (and fixed) in the formal description; and consequently deployed services are still prone to these kinds of errors.

On the other hand, SSDL fully describes the state space of a composed service as well as the sequence of service interactions (message exchanges) required to reach each state. We can thus view a composed service whose description is given in SSDL as a state machine, with message exchanges providing the transitions between states, and states implicitly defined as points between these exchanges.

Starting from this premise, we developed an SSDL Protocol Execution Engine that bridges the aforementioned gap between description and implementation by directly executing the state machine defined by the formal description. For each message exchange observed, the machine's state is advanced to the state specified in the description. Then, an action tied to this state can be invoked. This action leads to another message exchange, which in turn advances the state machine to the next state.

B.2.1. Implementation of SSDL Elements SSDL documents describe the state space in the form of a tree whose leaf nodes are <`msgref`> elements. These specify that the type of message referenced in their `ref` attribute be sent or received. Other elements define the order in which message exchanges in their subtrees need to occur. In respect to their influence on the state machine's behaviour, these fall into four classes:

Sequential Execution All child nodes must be executed sequentially, i.e. in the order they are given in by the SSDL document. A node with sequential execution is considered completed when all message exchanges required by its children have taken place. This class comprises the <sc>, <protocol> and <sequence> elements.

Parallel Execution The order in which child nodes are executed does not matter, but as with sequential execution all message exchanges must be completed before a parallel execution node is complete. This class is made up of the <parallel> element.

Branches Exactly one of the child elements must be completed. If the special <nothing> element is present, a branch may be skipped. I.e., execution of the <nothing> element implies that none of the other child nodes must be visited, hence none of the messages specified in their <msgref> descendants must be observed before the branch can be completed. The <choice> element is the only member of this class.

Loops All child nodes can be executed multiple times and in parallel, i.e. one loop need not be finished before the next starts. Loops are specified by the use of the <multiple> element. We do not support loops in our current implementation.

Our SSDL Protocol Execution Engine recursively visits and marks completed nodes according to the order of message exchanges observed and required by the SSDL description.

B.2.2. State-keeping in a stateless environment We developed the DealMaker service within an Axis environment. Web Services that use Axis RPC wrappers are inherently state-less: Every service invocation starts with a freshly-loaded executable. Services that need to keep their processing state between invocations have to save and restore the information necessary to do so to/from an external storage system (e.g. a database).

We distinguish two ways of keeping state. First, the state itself can be saved explicitly. With our engine implementation, this corresponds to saving the current state space tree, whose configuration of completed and uncompleted nodes represents the state machine's state. In an implementation, this provides a reasonably efficient method to reach the current state before continuing work.

Second, with a state machine that is driven solely by external input, state can also be kept by storing the input sequence that was encountered previous to reaching the current state. To restore state, the engine then steps through this sequence, ignoring actions tied to the states it traverses. In regard to reaching the current state after startup, this method is clearly less efficient than explicit state-keeping, because all steps of the machine have to be executed again before the actual action invoked can be taken. However, we favour it because (a) it is more flexible, and (b) helps to implement fault-tolerant applications. Higher flexibility results from the fact that the state machine description (the SSDL document) can be modified between service invocations, without necessarily invalidating any partially-completed processes. This is of particular importance with long-term processes such as that implemented by the DealMaker, where one process instance may be running for several months before all steps have been completed.

Furthermore, storing and re-reading the input sequence offers an obvious starting point for the application of fault-tolerance (FT) measures. N-Version Programming (NVP) as a means to improve the reliability of the SSDL Protocol Execution Engine itself illustrates this best. In short, NVP entails the use of several different implementations of the same component to eliminate errors introduced during the programming process. With input sequences stored and available in the same format to each version, individual implementers can concentrate on improving the core engine, and avoid inter-version dependencies in the state-keeping code.

B.2.3. Setting-specific implementation details We previously described the SSDL Process Execution Engine on an abstract level, considering message sequences as its input and unspecified 'actions' as what happens in the single states. In the following, we will point out several details of the implementation as part of the DealMaker service.

The general architecture is shown in Figure 10: Based on the state reported by the SSDL Process Execution Engine, the DealMaker invokes actions tied to each state. In addition, it offers facilities to keep the internal machine state consistent with the deal's real-world status.

At the moment, our implementation only performs one type of action: The user is presented with the state of the deal and with his options to progress it. Based on the machine state, we retrieve an explanatory, pre-generated HTML page from the database and deliver it to the user. While limited in its general applicability, this choice is adequate for the human-centric interactions that dominate our business case. In the future, invocations of services that help the user complete his deal could be tied to some states; e.g. a service that negotiates between schedules might help buyers and sellers set a date for the viewing of a property.

To keep the internal machine state consistent with the real-world status of the deal, the DealMaker must be aware of any message exchange that takes place between parties within the protocol. The most straight-forward way to achieve this is to implement the DealMaker as a message broker for all messages sent during the deal. However, this does not only involve privacy issues and performance considerations that may both hamper acceptance of the service,

but also reduces service flexibility by tying parties to one central entity.

We therefore chose to simply offer an interface for the user to notify the DealMaker that they have sent (or received) a specific type of message, i.e. that a message exchange has occurred. Note that the actual message contents are not of importance here; the DealMaker needs to know only the type of message that was sent. As we keep state solely through the sequence of messages encountered, the DealMaker only has to store the kind of message it received into the database.

At the core, both ways in which the DealMaker interacts with the outside world are implemented as Web Services using SOAP. External parties can both query and update the current deal state through the DealMaker. Access to these methods is possible in the standard Web Services fashion (i.e. by sending and receiving SOAP messages). The WS interface simplifies the creation of external services that make use of the DealMaker's functionality. In fact, our web interface, which hides these technical details from the human user, is implemented as an in-browser WS client sending/receiving SOAP messages.

B.3. Security and Trust Implementation

Messages between services carry security information, like identification or rights. In SOAR we express this kind of information using SAML authentication and attribute assertions. To make sure that the application layer does not need to understand about security information, the security solution is based on Axis handlers [3], that intercept in a transparent way SOAP messages exchanged among service instances. SAML assertion can be inserted and checked without interaction with applications (or, in fact, with users). The security implementation was done using the follow open source libraries: Apache XML Security–an implementation of XMLEncryption and XMLDigitalSignature [1]; WSS4J–a WS-Security implementation [2]; and OpenSAML–a SAML implementation [11].

B.4. Model Checking SSDL

SSDL was chosen as protocol and message exchange definition language to utilise the proving capabilities of the sequence constrains form of protocol specification. There is no automated tool support yet for model checking SSDL protocols, so we translated the specification by hand in the form of a π–calculus. In this way we were able to validate an earlier version of our protocol for correctness–the final version of our protocol should be checked again. Obviously, there is a great need for SSDL-related model-checking tool support to make it practical to check SSDL specification throughout the design phase, especially in light of the fact

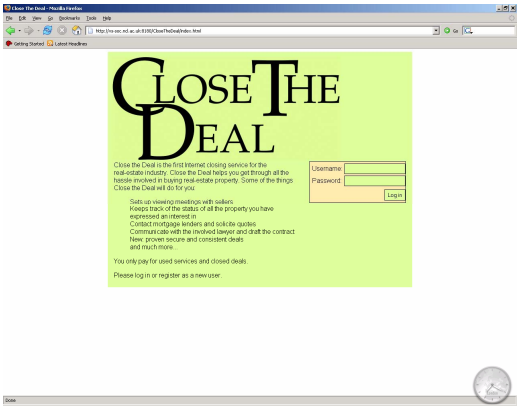


Figure 11. The demo welcome page.

that these specification may be altered at various time, for instance because of implementation decisions that alter or limit the protocol behaviour.

The need for tool support for SSDL is prevalent throughout the design phase. Existing tool support [18] creates typed representations of messages, supporting both C# and Visual basic code generation, as in addition validates the specification for consistency. However, the tools do not take the protocol framework of SSDL into consideration. Therefore, the most important novel implementation task in our system was the design and implementation of the SSDL protocol engine, which we described in detail in Section 5.2.

Because of time pressure, we were not able to validate the final version of our DealMaker service, as given in Figure 3. In SOAR interesting issues arise with respect to determining the state of multiple service instances concurrently. Each service instance itself runs a copy of the DealMaker protocol, but customers as well as the device run-time can be interested in multiple service instances at the same time. For instance, in the demo, we would like to check that at most one of the instances of each customer gets into the final phase–otherwise, the buyer would end up with multiple properties. Or, we would like to make sure real-estate properties are not sold twice, an even harder problem because it not only goes across multiple instance, but also multiple customer (i.e., buyers). Other states defined across multiple service instances could be thought of, and it would therefore be of interest to identify ways to express states of interests and execute the model checking in efficient manner across multiple instances at once. We have had to leave this for further research.

C. The Demonstration Web Site

Note that this description might be subject to change in some of the specifics.

In this section we detail the intend behind the demonstration web site (see Figure 11), and provide a manual of the actions one can execute on the web site. The focus of the web site is on demonstrating the viability of the technologies we applied–we have made no concerted effort to make a product-quality web site. It should be noted that the web site has mostly been tested for the Firefox browser–we strongly suggest one to use the Firefox browser, since we cannot provide any guarantee about correct operation for other browsers.

The purpose of the web interface is to demonstrate how a potential buyer would utilise the 'Close The Deal' real estate closing service to help with the process of buying real estate property. In what follows, we use 'user', 'buyer' and 'customer' to mean the same thing: the person who logged into the web site. The fact that we focus on the buyer has one immediate consequence: the user interface is limited to a view for the buyer, even though the service design support activities of other participants as well (the seller in particular). It should also be noted that the buyer is not aware of the fact that it uses a service-oriented computing architecture–on the contrary, this remains hidden. The only way in which a buyer would notice that the supporting implementation uses advanced technologies is through the advanced functionality the web site delivers.

Note that every time you log in to the web site, you start in the same process state!

A buyer that uses the Close The Deal web site needs to do two administrative steps (or one, in case of a returning customer), and then is guided through the process of closing real estate deals, for as many deals as the user desires. The two administrative steps are:

1. *Logging in.* Every time a user enters the web site, he/she needs to log in. This can all be implemented by standard means. We only use it to limit concurrent and ill-fated access to the web site, and support only one user, named `close` with password `thedeal`. Please use this account to view our web site.
2. *Provide buyer details.* A regular form needs to be filled out with information about the buyer. The interesting aspect of this is that one can fill in data that corresponds to a service instance in the SOAR architecture. For instance, one chooses a lawyer from a list of lawyer, each of which is represented by a service instance in our system. Note again that the user does not know it selects a service instance–the user only could realise the efficiency of the architecture when he/she realises there is no need to fill in any additional information about the lawyer (or other participants) after selecting it.

The interesting aspect for a customer comes when selecting houses in which it is interested. The user therefore must construct a *deal*, which the service then tracks. To construct a deal, the customer browses potential offers, and selects the one it is interested in. As a consequence of this set-up, there are two 'deal-making actions' a customer can select:

1. *Browse and select real-estate properties.* When the customer clicks the `Browse Offers` button, the browser displays a list of properties, of which one at a time can be selected. Selection of a property is simply done by clicking on the hyperlink located with the property.
2. *Check progress of the deal.* When the customer clicks the `Check Status` button, it provides the list of deals in progress. Important in this list is the *status* of each deal: how far along is the customer in closing the deal. Each deal goes through the steps of the business process, but the user is continuously involved in providing feedback on whether and how a next step needs to be carried out. The start state of out web site provides the status `initiate`. Once you click that, you are asked if you want to start the mortgage application, for the amount given in your profile or the asked price, which ever mount is larger (which, unbeknown to the buyer, is a property of a service representing the buyer). If you agree, the Close The Deal service will request mortgage lenders if they want to provide a mortgage. In the demo, you will simply have to wait for the process to finish (which in real life could take hours or even days), so essentially the demo stops here. However, this clearly demonstrates how the buyer interacts with the Close The Deal service to help getting through the process of buying a house. Follow-up steps, which we in fact also implemented the machinery for, include arranging a viewing meeting, contacting a surveyor and drafting a contract with a lawyer, along the lines of the protocol in Figure 3.

D. Reflection

The complete scenario and work described in this report was conceived, designed and developed within an eleven weeks time span. We briefly want to reflect on the main challenges we faced during the project: conception of the real-estate scenario, and the geographic distribution of the team members. We also review the business as well as research opportunities sprouting from the reported work.

D.1. Conception of SOAR

The open-ended nature of the contest brought a challenge as well as an opportunity to the work we would

be able to do. The choice of application area (real-estate) was partly motivated by existing contacts with the PISCES real-estate standardisation organisation located in Newcastle. However, the scenarios, business case and usage models were all developed from scratch. We have tried to utilise the industrial experience of the senior team member in creating business models for SOAR and SOAR services, but were mostly motivated by the technical questions 'why services' and 'how to do services'. In terms of time, the scenario conception took 40 percent, the design 20 percent, and the implementation 40 percent, but obviously the boundaries between these phases are loosely defined. It would be of great interest to us to continue some of the technical work, and a future opportunity to work on the SOAR architecture without a lengthy phase of conceiving scenarios would give us an opportunity to show even more exciting technical results. The sound technical skills and deeply developed intuition for service-oriented computing that is present in the industry advisors within our team was leveraged heavily, and is at the heart of the work on SSDL and the associated protocol execution engine. This provides us with some unique technologies that we feel are very much worth pursuing further within the SOAR context.

D.2. Geographic Distribution of Team

Our team was assembled solely for this contest. Timing-wise, the lucky opportunity arose for the people involved to spend time on the contest (in varying amounts). We participated with the objective to use the contest as a learning experience, as well as a source of fun by working as a team toward a common goal with a competitive element. In spirit with the openness associated with service-oriented computing, our team is globally distributed: the six team members are located in five different countries, four different continents. There are two industrial advisors (Jim and Savas) with a lot of experience in service-oriented software–these people also developed SSDL. Aad has close to a decade of industry experience and brings in some domain knowledge through contacts with the real-estate standardisation organisation PISCES [15]. Emerson is a PhD student specialising in security, Chris a PhD student specialising in software for distributed decision-making, and Philipp a master student who has published on web service reliability. The latter three implemented the system.

The distributed nature of the team provided obvious communication challenges we had to learn to deal with. From our industry experiences, we were aware of the challenge in starting new projects when a team is geographically dispersed, and it took us quite some effort to find a good way of producing as a team. For the purpose of the contest, we travelled to create periods in which all core members of the team were in Newcastle to work full-time on the project. This provided excellent results, and to our judgement periods of geographic collocation are a prerequisite for a team development project that starts from scratch. It did become clear, however, that technologically we still have a long way to go before Internet communication tools effectively support (in cheap and robust way) the kind of communication and interaction required for a high-pace concentrated team effort like ours.

D.3. What is Next?

It would be exciting to continue the work we started for this contest. There are opportunities to go after the business ideas presented in this report, and there exist opportunities to influence the standardisation bodies. From a technical perspective, the connection of human interaction and web services has been an eye opener, and is of interest to pursue further. The implementation ideas that relate to the hosting of many personalised service instances in an efficient manner need to be explored further. Possibly design patterns can eventually be derived from our solutions. Finally, we have gained considerable insight in the working of SSDL, which we would like to utilise to further improve that technology. This also includes improving the abilities to proof correctness of protocols with respect to concurrent service instances.