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

This paper introduces 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 [1].

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., [10]) 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 emerging [5, 7, 9, 11] that support the interaction between various players (and the software packages they use), including real estate agents and mortgage lenders ([14], see for more details [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 ensure privacy, identity and validity of user claims.

This paper describes ten weeks of work for the 2006 services computing contest of the International Conference on Web Services, 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

* 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, preineck@informatik.hu-berlin.de.

§ Chris Smith and Aad van Moorsel are with Newcastle University, 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.

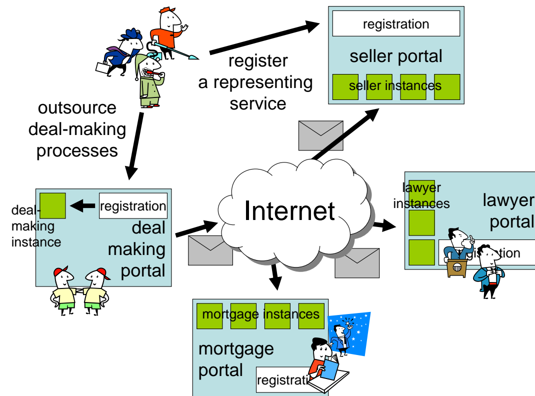

Figure 1. SOAR services landscape.

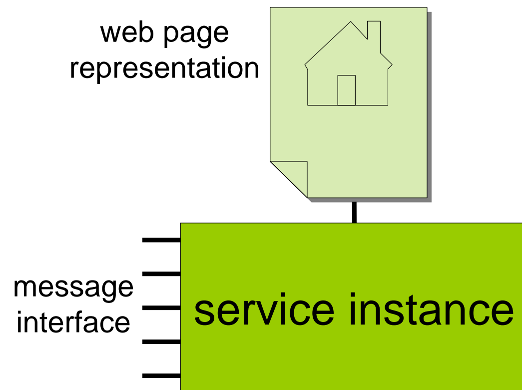

Figure 2. Service: message interfaces and web page representation.

diverse real-estate industry (Section 2).

- 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 3.1).
- we embedded a security solution within SOAR to guarantee privacy, identity and to 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 ([1]).
- 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, an associated technical report [2] provides 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 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 ac-

```
<?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.

cept parameterisation referring to other service instances. For example, 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 [2], 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 exploit SOAR. In [2] we discuss them in increasing order of complexity. There we also discuss the business case behind such services as well as behind SOAR itself.

3. Trust and Security Architecture

3.1. 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 [4]. 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

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <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">
3    <sc:sc>
4      <!--Parties In Mortgage Organise Protocol-->
5      <sc:participant name="Buyer" />
6      <sc:participant name="MortgageLender" />
7      <!--Message Exchange Protocol-->
8      <sc:protocol name="MortgageOrganiseProtocol">
9        <sc:sequence>
10         <sc:choice>
11           <sc:sequence>
12             <ssdl:msgref ref="msgs:MortgageRequestSubmission" direction="in" sc:participant="
                   Buyer" />
13             <ssdl:msgref ref="msgs:MortgageRequestTemplate" direction="out" sc:participant="
                   MortgageLender" />
14             <ssdl:msgref ref="msgs:MortgageRequestCompletedTemplate" direction="in"
                   sc:participant="Buyer" />
15             <sc:choice>
16               <sc:sequence>
17                 <ssdl:msgref ref="msgs:MortgageRequestAccepted" direction="out" sc:participant=
                       "MortgageLender" />
18               </sc:sequence>
19               <sc:sequence>
20                 <ssdl:msgref ref="msgs:MortgageRequestRejected" direction="out" sc:participant=
                       "MortgageLender" />
21               </sc:sequence>
22             </sc:choice>
23           </sc:sequence>
24           <sc:nothing />
25         </sc:choice>
26       </sc:sequence>
27     </sc:protocol>
28   </sc:sc>
29 </ssdl:protocol>
```

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

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 an SSDL specification. It has two participants involved, the buyer and the mortgage lender. When 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 initiates next steps as appropriate. The SSDL protocol execution engine is further discussed in Section 5.2.

The SOAR architecture requires solutions for common security issues such as authentication, privacy, etc., which we discuss in Section 3.2. 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 3.3.

3.2. Authorization, Confidentiality and Integrity

The communication among services and between services and web users is done using SSL, providing basic security properties such as confidentiality and integrity. The assumption is that all service providers have acquired X.509 certificates, 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 [6] to provide identity as well as authenticity in message exchanges. The authorization is done by a role-based access control [3] 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.

```
<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.

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.

3.3. Trusted Claims

In SOAR, individual participants could make unsubstantiated claims about ownership of properties, etc. In real life we can often easily enhance trust in such claims (such as ownership of a house) by paying a personal visit or searching 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 judgment calls about the validity of a participant claim, possibly simply because no humans are available with the right expertise. To protect SOAR from illegitimate usage, we use a trusted third party 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 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 [8], and we briefly summarise the main points of this approach to correctness validation. First we introduce SSDL, closely following [8, 12].

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-focused, 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 artifacts. 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 con-

straint 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 messages assumed by a receiver are not part of the protocol of the assumed sender.

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 [2].

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 [2].

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 instances 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. 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.

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

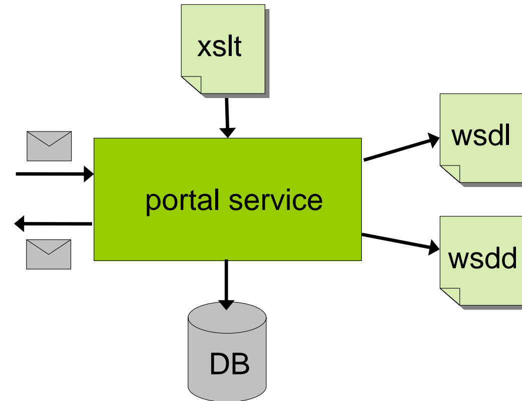

Figure 6. Service instantiation process.

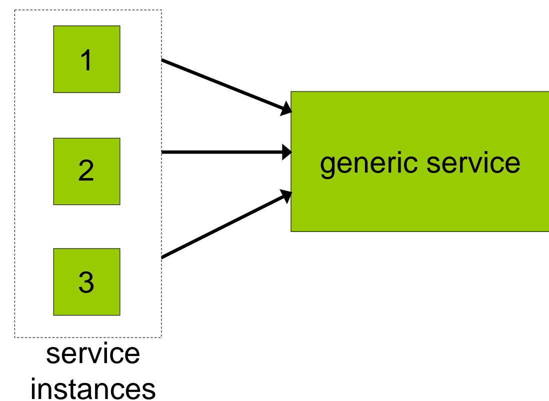

Figure 7. Service deployment process.

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 [13] 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 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 dealmaking 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.

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

Figure 8. Sample service instance deployment file.

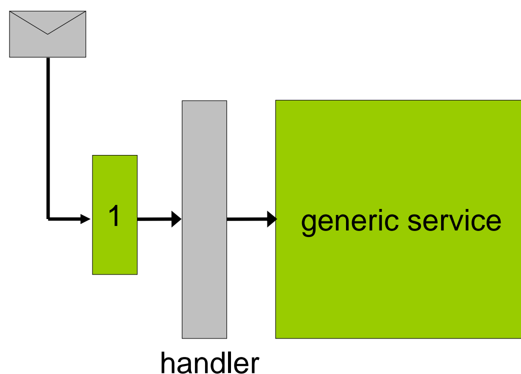

Figure 9. Service invocation process.

5.1.2. Service Deployment The final output of the above instantiation process is the web service deployment descriptor (WSDD) document. It is within this WSDD 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.

5.1.3. Service Invocation Deployment in the way described above 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 the same as 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, 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 semantics of the different elements. How this is done is discussed in detail in [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 requests so, based on the machine state, the DealMaker

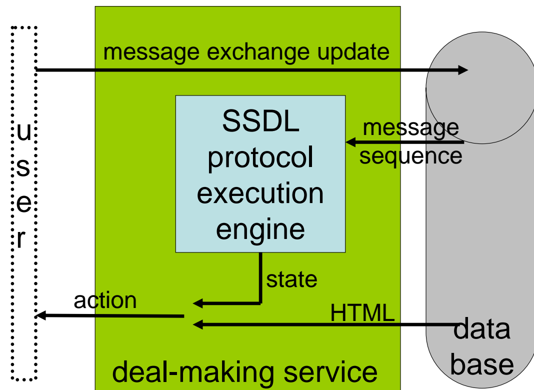

Figure 10. The SSDL Protocol Execution Engine within the DealMaker service.

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 [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 (discussed further in the full report [2]). Particularly novel is the use of formal proof techniques to demonstrate the correctness of our protocols, the solution for trusted claims, and the implementation of an SSDL protocol engine. Of interest is also the systemic inclusion of web interaction capabilities in SOAR web services, to leverage human skills and input within the process of automation.

References

[1] E. de Mello, S. Parastatidis, P. Reinecke, C. Smith, A. van Moorsel, and J. Webber. Demo of SOAR: Close the Deal. Temporarily available at `http://vs-soc.ncl.ac.uk:8180/CloseTheDeal/index.html` (account: `close`, password: `thedeal`), 2006.

[2] E. de Mello, S. Parastatidis, P. Reinecke, C. Smith, A. van Moorsel, and J. Webber. Secure and provable service support for human-intensive real-estate processes. Technical Report CS-TR: 960, School of Computing Science, University of Newcastle, May 2006.

[3] 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.

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

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

[6] 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.

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

[8] 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.

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

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

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

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

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

[14] xwebservices.com. Xweb1003. `http://www.xwebservices.com/Web_Services/XWeb1003`, 2006.