Computer and Information Science; Vol. 12, No. 2; 2019
ISSN 1913-8989 E-ISSN 1913-8997
Published by Canadian Center of Science and Education

Service Packaging: A Pattern Based Approach Towards Service Delivery

Muhammad Adeel Talib[1], Muhammad Nabeel Talib[2] & Madiha Akhtar[2]

[1] Genix Ventures Pty Ltd., Melbourne, Australia

[2] PNG University of Technology, Lae Papua New Guinea

Correspondence: Muhammad Nabeel Talib, PNG University of Technology, Lae Papua New Guinea. E-mail: atalib@genixventures.com, muhammad.talib@pnguot.ac.pg

Received: February 3, 2019 Accepted: February 18, 2019 Online Published: March 25, 2019

doi:10.5539/cis. v12n2p14 URL: https://doi.org/10.5539/cis.v12n2p14

Abstract

Authentication, authorization, billing and monitoring are all common service delivery functions that are generally required to be added on to core business services in order for them to be delivered online commercially. Extending core services with these service delivery functions requires considerable effort if implemented ground-up and can be subject to limitations if outsourced to a service broker or a conventional middleware platform. Because of the ubiquitous nature of these service delivery functions, we see them as reusable patterns for service delivery. In this paper we have introduce an approach to implementing and applying these patterns in business to consumer e-commerce. We name the approach Service Packaging. Through the approach, generic implementations (or service packages) of the various service delivery patterns can be incrementally applied to core services, thus enabling flexible and systematic service delivery. A core service, regardless of its business domain does not require any structural or behavioral modifications in order to conform to a specific service delivery requirement and hence can be used out of the box. We also present a prototype middleware platform for the design-time modeling and implementation of service packages as well as their runtime execution.

Keywords: service packaging, service delivery functions, role oriented adaptive design

1. Introduction

With the maturity of web service tools, creating and exposing backend functionality as web services has become relatively easy. In a commercial setting, however, there is more to simply exposing backend functionality as a service. Services have to be secured, consumers have to be authentified and authorized for the usage; billing mechanisms have to be in place; monitoring and metering should be enabled; and proper logs have to be maintained. All these service delivery functions have to be 'packaged' with the core service.

Service providers entering e-service arena face two challenges in implementing service delivery functions. a) weak infrastructure and capacity b) they simply don't want to spend extra energy on other than their main competencies. They are looking for ways to delegate these delivery functions to third party mediators. On the other hand, service providers understand that fierce market competition and rapid evolution in the technology would pressurize them to fine-tune their existing service offering or to incorporate new offerings on the fly, and hence they would like stay in control of the delivery chain. These two challenges put service providers in a dilemma when it comes to deciding the strategy to opt because generally delegation means lesser control.

Recently, cloud based service brokers and API management tools have caught the attention of service providers who are trying to find ways to delegate the provisioning of these service delivery functions to external mediators (Li et al., 2009). However, while these platforms provide service delivery functions in some form and promise high scalability in terms of infrastructure resources, they do suffer from the limitations of a rigid service delivery model (Weber et al., 2009). The service providers have to abide by the static built-in service delivery functions or the configurations offered by these cloud platforms which may not conform to their dynamic business rules. This reliance on cloud platforms vendors to supply appropriate customizations to meet business needs results in the loss of control over the business' service delivery chain (Barros and Dumas, 2006) and eventually loss in revenue in the long term. Middleware technologies such as ESBs have been evolving over time to include a wide variety of management functions across a whole portfolio of services. These provide some flexibility in terms of the control

over the service delivery functions but are focused on EAI (Enterprise Application Integration) and are generally heavy-weight and costly. Gartner (Thompson et al., 2013) has reported that cost remains the main challenge that most of the modern ESB vendors face in selling their products.

The necessity is to use a balanced service delivery solution where vendor lock-in can be avoided with a light-weight service delivery platform which is responsive to on-the-fly alterations. To overcome the shortcomings of the above approaches we have introduced a 'service packaging' framework. We see service delivery functions e.g. logging usage statistics, authentication, charging users for service usage, monitoring service response times, etc. as commonly recurring service delivery patterns. These patterns are implemented as reusable 'service packages' or templates. Furthermore, these service packages (or reusable implementations of service deliver patterns) can be incrementally added to a core service as and when required, enhancing the service delivery chain that comprises of incremental functional enhancements (see Fig. 1).

Furthermore, as a proof of concept we have implemented a prototype by extending an existing adaptable service composition framework called ROAD (Colman et al., 2007). The prototype allows for the design-time modeling of service packages and runtime execution of packaged services.

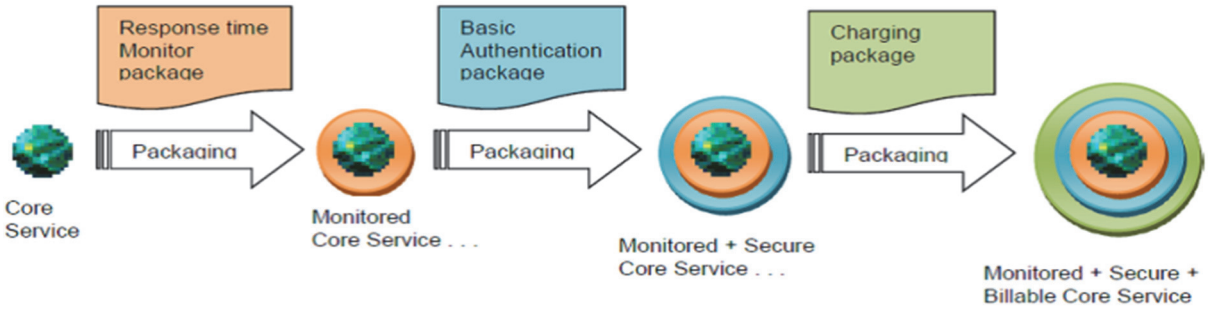

Figure 1. Example augmentation of a core service with delivery functions in the service delivery chain

The rest of the paper is organized as follows. In Section 2, we elaborate on the methodology. We then introduce the ROAD framework in Section 3 which qualifies for a desirable service delivery solution. In Section 4 we describe a case study in which a service provider wishes to expose a backend function as an online service and incrementally enable it with delivery functions. To validate the approach, in Section 5, we demonstrate the implementation of the case study using the ROAD platform. Related works are presented in Section 6 followed by conclusions in Section 7.

2. The Approach – Service Packaging

The steps involved and the artifacts produced during the service packaging process are explained below.

Step1- Identify a service delivery pattern

A service delivery pattern describes a specific recurrent functional or non-functional business requirement (e.g. pre-paid charging, authentication, authorization, monitoring, etc.) that service providers need to add-on to their core functional services. While these patterns are common in today's e-commerce, the challenge is to generalize their implementation so that it can be reused in multiple applications across various domains. Therefore, the first step in service packaging process is to analyze if: i) a service delivery function is a value-adding function that is frequently required by service providers; and ii) its implementation can be generalized.

For the explanation of this step, let us take an example of a basic authentication pattern. This example provides the context for the rest of the steps of the approach. The definition of the basic authentication service delivery pattern is as follows:

Name: Message level basic authentication

Motivation: Service providers may need to authenticate the consumers before access is granted to the service. The authentication is a pre-requisite for the subsequent charging / billing if required or it can just be for auditing purposes.

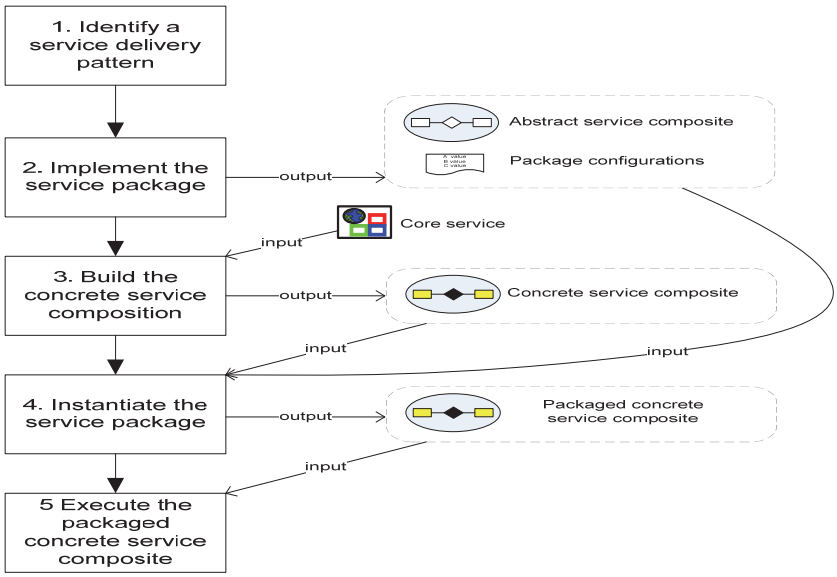

Figure 2. Service packaging steps

Solution: There are three basic steps in the implementation of this pattern:

a) The service consumer's username and password is extracted from the request message.
b) Match them with the user credentials in an identity database.
c) If found, the user is allowed access to the service. Otherwise, a fault message is generated and returned.

Configurable Parameters: Binding to the external identity database and its implementation.

Consequence: Only registered users with valid username / password are allowed to access the service. Fault is generated and replied otherwise.

Figure 3 below illustrates the design requirements of a basic authentication service delivery pattern.

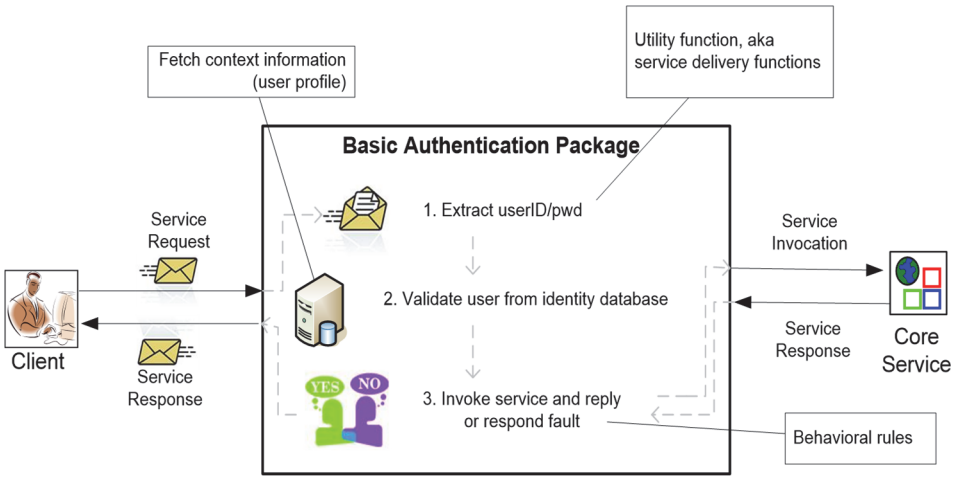

Figure 3. A generic Basic Authentication pattern requirements

Step 2 - Implement the Service Package

Once a service delivery pattern is identified the next step is to implement it as an abstract service composition template (or service package) that bundles the pattern-specific functionality in it. The service package may have placeholders for domain-specific configurations that will be used when the package is being applied to a specific domain (also known as package instantiation). The service package itself is independent of the domain knowledge. The service price, the allowable response time, and the time period during which access to the service is denied are example of such configurations.

Step 3 - Build the Concrete Service Composition

The core service is the actual business service on to which the service delivery functions need to be added. In order to add service delivery functions onto a core service, the service will have to be first encapsulated within a concrete service composition. Concrete here means that the composition should be executable with a valid end-point address. In its simplest form, the concrete service composite shall contain the core service and a proxy client role. The core service role is bound to the actual functioning core service (player) through a concrete endpoint address.

A concrete composition is the basic building block of the service packaging approach. As opposed to applying the service package directly to a core service, applying it to a composition allows the service packages to be applied at the interaction level, i.e. the interaction between the client and the service. For example, the rules governing the access restrictions to a service are dependent on both the client and the service and hence defined at the interaction level.

Step 4 - Instantiate the service package

Instantiation is the process of applying a service package to a concrete service composite so that the service delivery functions implemented in the service package can be enabled for the core service encapsulated in the concrete composite. This is achieved through merging the service package into the concrete composite producing a concrete deployable composition artifact.

Step 5 - Execute the packaged concrete service composite

The packaged service composite that comes out of the instantiation step is also a concrete service composite that includes the structure and behavior of both the core service composite and the domain-independent service package. This composition is then deployed and enacted in a runtime execution engine (the next section provides more details of the execution engine). The packaged service composite can recursively act as a concrete service composite that can be further packaged with additional service packages.

3. ROAD Platform Prototype

For the implementation of the service packaging approach we use the Role-Oriented Adaptive Design (ROAD) platform (Colman, 2006). ROAD is built on top of open-source technologies including JBoss Drools (Bali, 2009), JAXB, Axis2, Tomcat, Eclipse and EMF/GMF.

It consists primarily of three main modules as shown in Figure 4 below:

An eclipse based IDE called ‘ROAD Designer’ for the design time modelling of service composites;

An Axis2 based server called ‘ROAD Web Container’ for deploying the service composites; and

A runtime engine called ‘ROAD Engine’ to control the message routing the service composites.

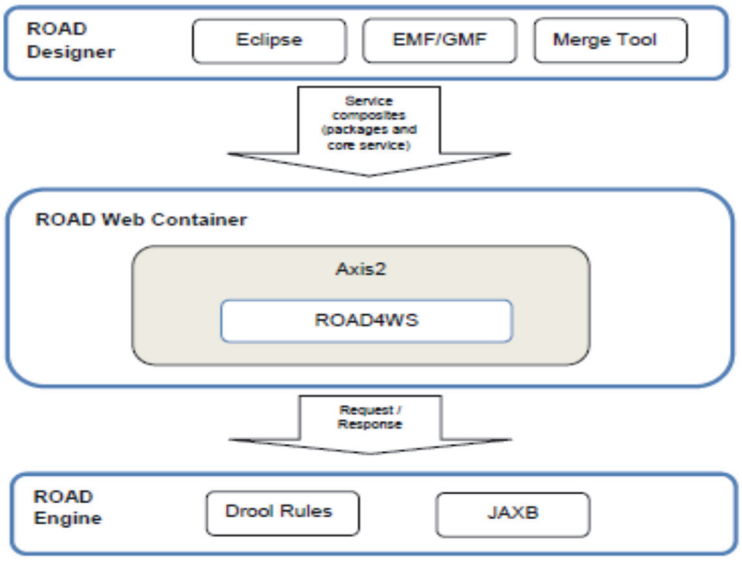

Figure 4. High level ROAD architecture

A brief description of ROAD components is provided next.

3.1 ROAD Designer

ROAD Designer is an eclipse-based modeling IDE implemented using EMF/GMF technologies. GMF provides tool support for generating graphical editors of domain models. Here the domain model is the ROAD model. A screenshot from the ROAD Designer IDE is shown in Figure 5.

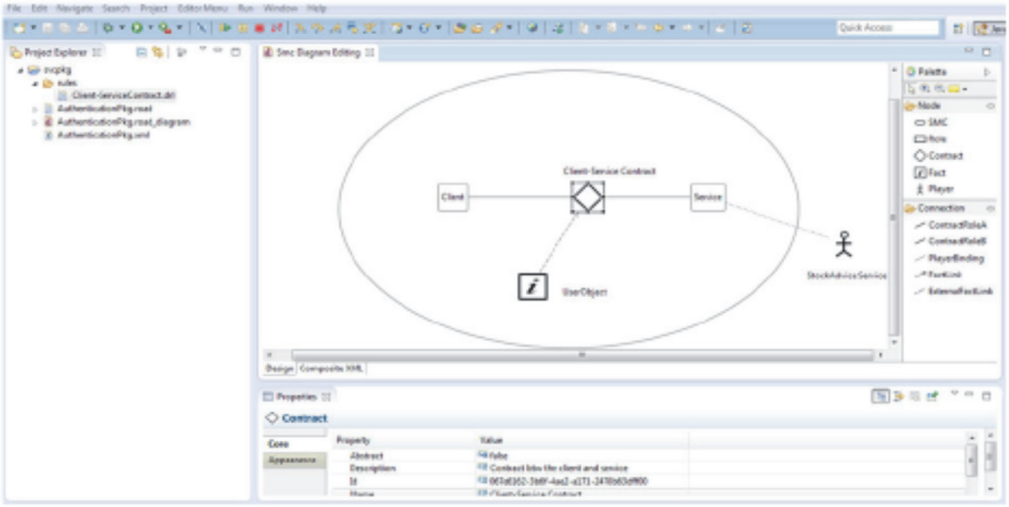

Figure 5. Screenshot of ROAD Designer IDE

ROAD provides utility functions such as an event registration function (that allows the rule engine to trigger off the rules based on a condition), message extraction functions (that gets the value of an XML tag from the SOAP header), returning the system time (for analyzing response time), etc. The users can use them out of the box when implementing the service packages and additionally can implement new functions as required.

A merge tool has been implemented for instantiating the service package. The tool takes in the concrete service composite, the service package composite, rule and configuration files and generates a packaged concrete service composite. This packaged service composite is then deployed for execution in the ROAD Web Container.

3.2 ROAD Web Container

The container contains ROAD4WS which is the Web service layer of the platform that extends Axis2. A hot deployment mechanism is introduced that picks up any service composition configuration (XML and rule files) dropped in the installation folder at runtime. The container then creates the appropriate stubs and skeletons for the services. All messages to the core service are routed through the ROAD Engine.

The ROAD4WS supports both push and pull message exchange patterns with one-way asynchronous or two-way synchronous communications. It provides message handlers that intercept the messages bound to a role, sets message context properties such as name of the operation invoked and hands it over to the ROAD Engine component which then executes the rules to control the behavior of the composite.

3.3 ROAD Engine

The ROAD Engine is bundled with the JBoss Drools rule engine (Bali, 2009). ROAD4WS passes the composite XML and rule files to the ROAD Engine for building an in-memory runtime model of the composite using JAXB. The engine listens to the messaging events from the ROAD4WS and in-turn pushes the event to the Drools engine that then triggers necessary actions as per the service composite rules.

A runtime management interface (King and Colman 2009) is also provided by the ROAD Engine through which runtime adaptations such as adding/deleting/updating rules, roles, players, etc. can be triggered.

4. Implementation of Service Packages

ROAD provides a domain specific model to implement service packages. The main constituents of the ROAD model are: A Self-managed Composite, a collection of Roles and Contracts that bind two roles together. Roles define a function that is performed by an external entity or Player, while Contracts consist of Configurations and Rules that define the interaction protocols between the two roles. A contract Mediator is a special type of role that provides contextual information used to evaluate the rules associated with the various contracts. Figure 6 represents

a self-managed composite and displays the other various components.

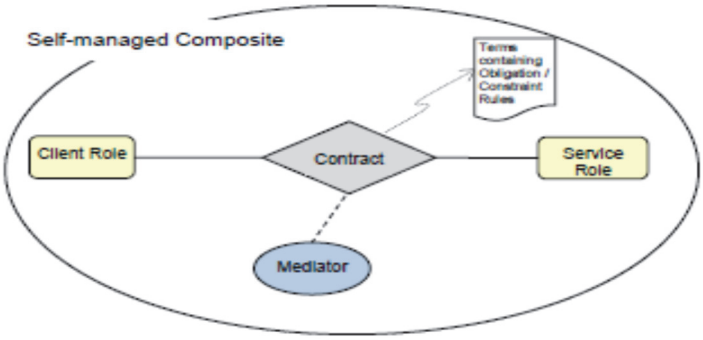

Figure 6. ROAD Conceptual Model (derived from Koutsopoulou et al., 2007) Design artifacts

Below we explain how a self-managed composite can be used to implement a service package.

1. Service Package

To implement a service delivery pattern in ROAD, an abstract service composite is built in which there are two abstract roles corresponding to a service and a client respectively. We model them as abstract because we want them to be decoupled from their actual players (or services). In ROAD, the client role acts as a proxy that provides the necessary stubs for service invocations.

2. Rules and configurations

Rules are defined to control the application logic and to record domain-specific metadata. Placeholders are used in definition of a ROAD contract. The values of the placeholders are provided through a configuration file. Rules and configurations are stored in a ROAD contract. The contract triggers the rules. In ROAD a special role called mediator is tied to a contract and is responsible for providing data required by the contract for rule engineering or for validating the data that passes through a contract. In the case of a basic authentication pattern, a mediator can be used to fetch user objects from an external data source. The user objects obtained through the mediator can then be directly utilized in the rule definitions providing more control to the developer implementing the rules.

3. Concrete service composition

A concrete service composition can be built with ROAD using the same elements as that of an abstract service package with the only difference that a concrete composition may have concrete roles bound to actual players having endpoint addresses.

4. Packaged concrete service composition

Packaging a concrete service composition is the most complex step for the approach and hence explained in detail in this section. With the ROAD Merge tool, users can apply the service package to a concrete service composite. To set the domain-specific parameters, the instantiation process also takes in the configuration file which provides the mapping of the placeholders in the service package to the actual domain-specific values e.g. the value of service price is set to 10 cents, valid response time is set to 5 seconds, etc. Figure 7 below diagrammatically shows the inputs to the instantiation process and its output.

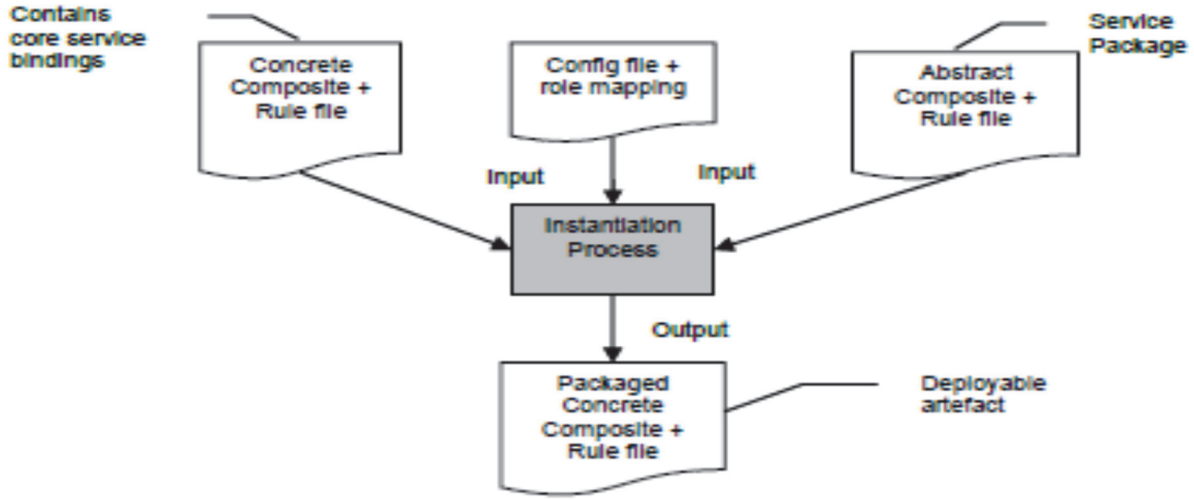

Figure 7. Input / Output of the package instantiation process

Given the inputs and outputs, we now explain how the instantiation work.

Definition 1:

A Service Composite SC is a tuple < R, C, P, A> where,

R is a set of roles such that $R = \{r_1 \ldots r_n \mid n \geq 2\}$,

C is a set of contracts such that $C = \{c_1 \ldots c_n \mid n \geq 1\}$,

P is a set of players such that $P = \{p_1 \ldots p_n \mid n \geq 0$ and $(p \rightarrow r \in R$ or $p \rightarrow a \in A)\}$

A is a set of mediators such that $A = \{a_1 \ldots a_n \mid n \geq 0\}$, and

A contract $c \in C$ is a tuple $< r_i, r_j, T, F, a_c >$ where,

r_i and r_j are references to the two roles that the contract c binds. Also $r_i, r_j \in R \ \& \ i \neq j$

T is a set of Terms such that $T = \{t_1 \ldots t_n \mid n \geq 1\}$

F is the set of rule files such that $F = \{f_1 \ldots f_n \mid n \geq 0\}$

a_c is an optional the reference to the mediator attached to the contract (we only allow one mediator per contract)

Definition 2:

A Concrete Service Composite SC_{conc} is a service composite that has the same semantics of the general SC in definition 1 with the exception that all elements would be considered concrete.

$SC_{conc} = < R_{conc}, C_{conc}, P_{conc}, A_{conc} >$ where,

R_{conc} and C_{conc} will only contain concrete roles and contracts respectively.

As an example, the concrete service composite in figure 8, the 3 roles r_A, r_B, r_C and the 2 contracts c_{AB} and c_{AC} are all concrete. Note in the figure that the concrete roles and contracts are drawn with solid line boundaries. Abstract roles and contracts are drawn with dotted line boundaries.

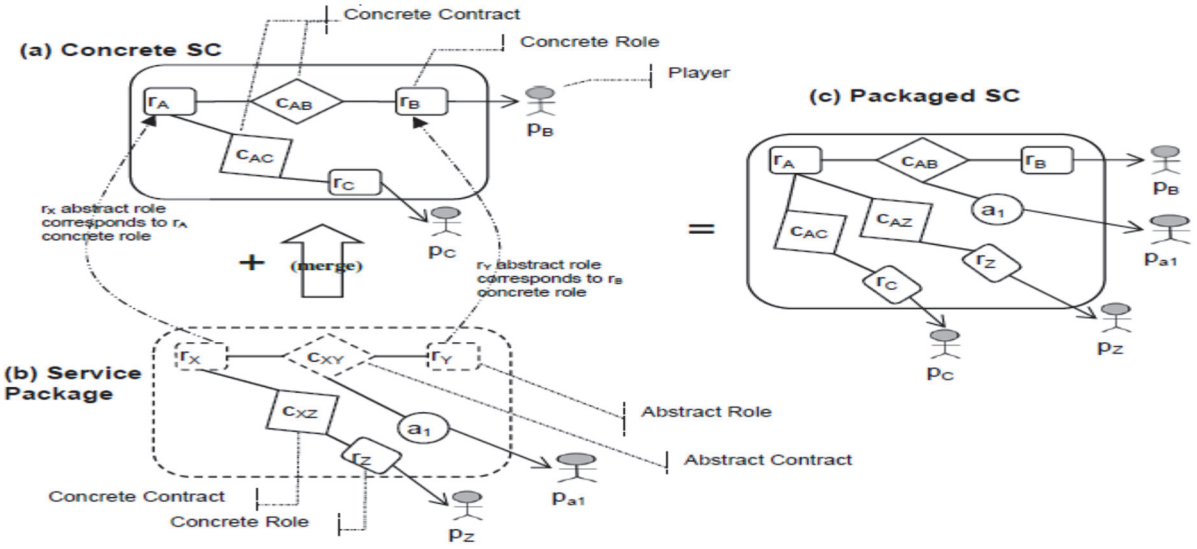

Figure 8. Package Instantiation (a)A concrete SC to which a service package is to be applied;(b)A service package;(c)The package service composite which is the superimposition of the service package into the concrete service composite

Definition 3:

A Service Package SC_{pkg} is a service composition that has the same semantics of the general service composite in definition 1 with the exception that there must be exactly 2 abstract roles r^{A}_{abs} and r^{B}_{abs} bound by 1 abstract contract c_{abs} in the service package. That is,

$SC_{pkg} = < R_{pkg}, C_{pkg}, P_{pkg}, A_{pkg} >$ where,

$R_{pkg} = R_{conc} + r^{A}_{abs} + r^{B}_{abs}$ // R_{conc} because the service package may have concrete roles such as r_Z in Figure 8 but must have 2 abstract roles such as r_X and r_Y in Figure 8.

$C_{pkg} = C_{conc} + c_{abs}$ // C_{conc} because the service package may have concrete contracts such as c_{XZ} in Figure 8but must have one abstract contract such as c_{XY} in Figure 8.

$c_{abs} = < r^{A}_{abs}, r^{B}_{abs}, T_{abs}, F_{abs}, a_{abs} >$ where,

r^{A}_{abs} and r^{B}_{abs} are references to the two abstract roles that the contract c_{abs} binds.

Also, $r^{A}_{abs}, r^{B}_{abs} \in R_{pkg}$ & $r^{A}_{abs} \neq r^{B}_{abs}$

T_{abs} is a set of Terms such that $T_{abs} = \{t_1 \ldots t_n \mid n \geq 1\}$

F_{abs} is the set of rule files such that $F_{abs} = \{f_1 \ldots f_n \mid n \geq 0\}$

a_{abs} is an optional reference to the mediator attached to the abstract contract such as a_1 is Figure 8

P_{pkg} is a set of players such that $P_{pkg} = \{p_1 \ldots p_n \mid n \geq 0$ and $(p \rightarrow r \in R_{pkg}$ or $p \rightarrow a \in A_{pkg})\}$

A_{pkg} is a set of mediators such that $A_{pkg} = \{a_1 \ldots a_n \mid n \geq 0\}$

Merge Algorithm

Inputs: SC_{pkg} , SC_{conc} , ConfigParam file(s), role mappings i.e. $r^{A}_{abs} \rightarrow r^{A}_{conc}$ & $r^{B}_{abs} \rightarrow r^{B}_{conc}$.

Output: *Packaged* SC_{conc} with the specific package functionality superimposed / merged

// Obtain the concrete contract cconc in the Cconc from the role mappings. For example CAB in Figure 8.

for each c in Cconc {

if (c.ri = r'_{conc} and c.rj = r'_{conc}) {

cconc = c ;

```
            return ;
        }
    }
    // Iterate over each contract in SC_pkg and find the abstract contract that would map to the concrete contract in the
    SC_conc. For example C_XY.in  Figure 8 is the abstract contract.
    for each c in Cpkg {
        // If this is the abstract contract
        if ( c.ri = r'_abs  and c.rj = r'_abs  ) {
                    // Add its rule files to the corresponding Cconc in SCconc  for example merge CXY rule files to
    CAB.
            for each f in Fabs {
                        // First replace config param with values.
                f' = instantiateRuleFile(f, configParamFile)
                Cconc.F += f;
            }
                // Add all abstract contract terms to the corresponding Cconc in the concrete service composite.
                Cconc.T += Tabs;
            }
                // Add all its mediators to the corresponding C_conc in the concrete service composite.
                Cconc.A += A_abs ;
            }
        }
        // If this is not the abstract contract, check if it is bound to an abstract role (e.g. C_XZ is a concrete contract
        bound to abstract role r_X). If so, change this binding to that of the corresponding concrete role (r_A) and add
        the contract (C_AZ) to C_conc.
        else if ( c.r_i =  r^A_abs  and c.r_j !=  r^B_abs  ) {
            c.r_i =  r^A_conc
            C_conc += c
        }
        else if ( c.r_i !=  r^A_abs  and c.r_j =  r^B_abs  ) {
            c.r_i =  r^A_conc
            C_conc += c
        }
        else {
            // Add all other contracts as is (e.g. C_AC).
            C_conc += c
        }
    }
    //Except  r^A_abs  &  r^B_abs  copy all other roles in Rpkg to Rconc
    for each r in Rpkg {
        if r !=  r^A_abs  and r !=  r^B_abs
            Rconc += r
    }
    // Merge player bindings.
    Pconc += Ppkg
```

// Merge mediators.
Aconc += Apkg

The rule file instantiation is a simple find-and-replace algorithm that replaces the parameter placeholders with their actual values

// instantiateRuleFile(*f*, *configParamFile*) subroutine

create key-value pair map (Θ) of the config params

parse each line *l* of *f*
if (*l* contains Θ.key)

replace Θ.key with Θ.value

5. Case Study-Trade Analytics

To demonstrate the potential of our service packaging approach, let us now take a case study of a stock exchange broker TradeAnalytics Pty Ltd that provides trading recommendations based on proprietary forecasting algorithms. The case study presented here covers both anticipated and unanticipated situations during the process of the company launching its online services incrementally.

Currently the company's stock recommendation service is offline; it wishes to offer the service into e-service marketplace to be used by online agents for the day-traders. In order to do that, TradeAnalytics wraps its proprietary forecasting algorithm in software and release it as a Web service called the StockAdvice service.

Scenario I

TradeAnalytics wants to launch the service initially free of charge for the public as a trial and enable monitoring to gauge the performance of the service in terms of response time. After 3 months, the service is only accessible to register users, still free of charge. User registration is done through the company's CRM system where the user records are stored.

Scenario II

After another 3 months, introduce a prepaid charging mechanism for registered users. A user will be required to maintain sufficient credit in their account to be able to use the service. At each service request, the user's account will be checked for sufficient balance and the amount equal to the service price (which may vary in future) will be deducted from the account balance after invocation. The company aims, to extend its clientele by providing options like post-paid charging, monthly/yearly unlimited usage subscription, and a set number of queries purchased in advance.

Scenario III:

Since the completion of the previous development phase, the core StockAdvice service has been up and running for some time. Amidst flourishing revenue from the new online service, TradeAnalytics decides to offer discounted price for off-peak hour service invocations. Also, it is noticed that a few service requests were being delayed due to the high processing demand of the backend forecasting algorithm. The company decides to offer discount rates if the response time was slower than a certain threshold. It was required that these parameters be configurable and also the company couldn't afford to have a service downtime. The company wanted these changes to be applied dynamically without shutting down the service at all.

5.1 Analysis

The scenarios represent a general trend in business marketing where organisations incrementally add value to their services. The StockAdvice service is the core business service of TradeAnalytics Pty Ltd. The company's strategy is to launch the service online and add on service delivery functions incrementally. First the service needs to be launched free of charge. Then a response time monitoring function needs to be implemented to monitor the service performance. Then the service should be enhanced to allow access to registered users only. Then a pricing mechanism is to be put in place. All these incremental value additions are service delivery functions and if a platform is available where these service delivery functions can be obtained out-of-the-box and applied on to the core service along with an enactment engine for service execution, then not only it would reduce the time to market but also will give TradeAnalytics Pty Ltd the flexibility to add value incrementally.

5.2 Case Study Implementation

With the help of the motivational scenarios presented above, we now elaborate how ROAD can be used to quickly develop reusable service packages and how the service packaging approach can be applied on a core service that

provides stock trading recommendations. In this section we explain how the company developers will go about implementing the solution.

Scenario I:

In the first phase of the project, response time measurement and authentication needs to be added on to the company's core *StockAdvice* service. The developers at TradeAnalytics take the following steps:

Step 1: Identify the service delivery pattern

The developers at TradeAnalytics understand that the service delivery functions required at each stage are patterns of service delivery and that these functions would be required for future offerings as well. They take the packaging approach and instead of hard-coding the logic into the middleware application layer, implements these patterns as reusable packages. For brevity, we have skipped the response time measurement package and have started from the second phase, i.e. authentication. The developer takes the following steps:

Step 2: Implement the service package

At design time, using the ROAD graphical designer (see Figure 9 for the screenshot of the tool), the TradeAnalytics developers start creating the authentication service package that would authenticate service requesters by looking up the registered client records in the company's Customer Relationship Management (CRM) system. Using the drag-n-drop feature of the ROAD Designer, the developers model the abstract service composite. Figure 9 represents the corresponding basic authentication design as a ROAD model.

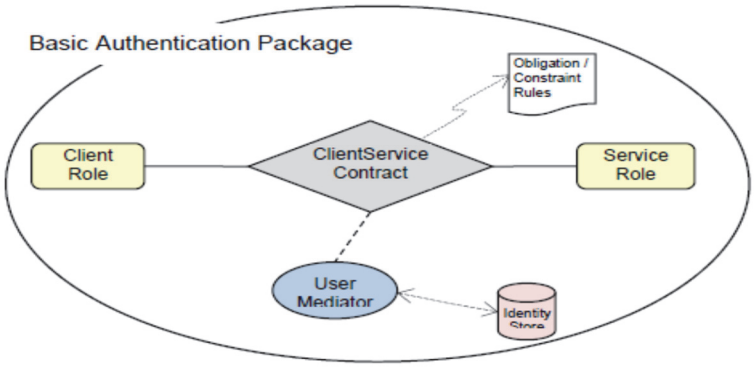

Figure 9. ROAD model for a basic authentication package

ClientRole and *ServiceRole* in Figure 9 are the two abstract roles and *ClientService-Contract* is the abstract contract that contains the terms and rule definitions that govern the interaction logic between the client and service abstract roles.

Rules are added on the contract by using Drools rule language (Bali, 2009). Listing 1 presents the authentication rule that is validating the service requestor. In line 3, the message received from the Client role is checked to see if it is destined for the correct core service. The *MessageRecievedEvent* is a ROAD function that registers the event of the message being received by the service composite from the client. *PARAM_FunctionalRequest* placeholder is used to model the actual service request. Line 4 iteratively stores *FactObjects* which are basically representations of the registered users fetched from an external data source through the *UserMediator* role which is a mediator implementation. *UserMediator* is placed on the contract that would be used to fetch user objects from the CRM. In lines 5 and 6, platform utility function *getValueFromHeader* is being used to obtain the username and password values from the message and we are validating if the username from the message header is equal to any user in the user repository *$fo*. This user repository contains all user accounts represented as facts in the system. These user objects (or facts) are loaded into the working memory through the *UserMediator*. The implementation of this mediator interface is explained in the next step. In line 8, the message is allowed to pass through if the username and password match.

```
1 rule "basic authentication"
2 when
3   $event : MessageRecievedEvent(operationName == "PARAM_FunctionalRequest")
4   $fo : FactObject()
5   eval(!($fo.getAttributes("name").equals(UtilityFunctions.
        getValueFromHeader("UserName"))))
6   eval(!($fo.getAttributes("pwd").equals(UtilityFunctions.
        getValueFromHeader("Password"))))
7 then
8   $event.setBlocked(true);
9 end
```

Listing 1. An authentication rule definition

The mediator is required to talk to TradeAnalytics' Customer Relationship Management system so that user objects (aka fact objects) can be loaded into the rule engine's working memory. To keep the implementation decoupled from the mediator role in the service package, the developers dynamically load the mediator implementation at runtime using the dependency injection mechanism. This allows the mediator role in the service package to hook into various domain implementations and it also keeps the service package intact if the same mediator implementation changes.

The service package contains a placeholder *PARAM_FunctionalRequest* in the authentication rule (Listing 1) to represent the name of the service operation in the concrete service composite. The developers create a property file and set the value for this parameter as *getAdvice* which is the actual method to be invoked. The developers will also specify the path to the fully qualified name of the *UserMediator* class. These configuration parameters are captured as key-value pairs in the property file as presented in Listing 2.

```
PARAM_FunctionalRequest = getAdvice
PARAM_UserMediator = au.swinburne.org.UserMediator
```

Listing 2. Contents of the configuration file for the basic authentication package

Once the package is designed, the package service composite is saved as an XML file. The rules will be written to a file with drl extension. See the Appendix for a sample authentication service package composite XML file. The package XML file, the rules file and the configuration file all are the artefacts generated as part of the service packaging approach outlined in the Section Design artefacts.

Step 3: Create the concrete service composite

The developers will then model the concrete service composition in a manner similar to implementing the service package (in the previous step). The difference being that in the concrete service composite, the roles created will be flagged as concrete and the service role will have a player bound to the actual core service. The functional interaction would also be defined in a concrete contract that binds the two concrete roles, i.e. a *getAdvice* operation that takes in a String *stock_symbol* and returns the response as a String shall be defined as a term. The concrete service composite is also saved as an XML file which forms the second artefact of the service packaging approach. In the context of the case study there are no rules associated with the concrete service composite although it may have domain-specific rules defined.

Step 4: Instantiate the service package

The developers will use the ROAD Merge Tool to provide the paths to the core service composite XML file, the abstract service composite XML file and the mappings between the abstract and concrete roles. After capturing these details the merge algorithm is executed that will merge together the concrete and abstract service compositions to produce one packaged composition in XML form with the associated rule files instantiated. This deployable packaged service composite is the last artefact produced in service packaging process which is then used for deployment.

Step 5: Execute the packaged concrete service composite

Once packaged, the service composite is deployed into the Axis2 container through a hot deployment mechanism,

i.e. by simply dropping the artefact files in the designated Axis2 server folder. The server exposes the role *Client* as Web services with WSDL interface having the *getAdvice()* method. The *StockAdvice* service acts as a proxy to the actual service and is hidden from public. At runtime, a client application (player) invokes the *getAdvice* operation on the public *Client* interface (role). The *getAdvice* request message is intercepted by the ROAD4WS handler, wrapped with the message context and handed over to the ROAD Engine which will then triggers the rules for user authentication before passing on the request message to the actual *StockAdvice* service. The same thing happens on the way back to the service requester.

Scenario II:

Step 6: Creating and applying the Pre-paid Charging package

After having security in place, TradeAnalytics wanted to put in a pre-paid charging mechanism. The developers repeat step 1 to 5 with the difference in the rules and the configuration parameters. The charging rules are presented in Listing 3 below. Here in, addition to the functional interaction placeholder, the other parameter is the service price. The prepaid charging package will then be applied to the *StockAdvice* service already packaged with the authentication package.

```
1  rule "check user balance"
2  when
3    $event : MessageReceivedEvent(operationName == "PARAM_FunctionalRequest")
4    $fo : FactObject()
5    eval($fo.getAttributes("name").equals(ServiceDeliveryFunctions.
         getValueFromHeader("UserName")))
6    eval((Integer)($fo.getAttributes("balance")) < PARAM_ServicePrice)
7  then
8   ServiceDeliveryFunctions.generateFault("Insufficient funds");
9  end
```

```
1  rule "deduct amount for service invocation"
2  when
3    $event : MessageReceivedEvent(operationName == "PARAM_FunctionalRequest",
                 syncResponse == true)
4    $fo : FactObject()
5    eval($fo.getAttributes("name").equals(ServiceDeliveryFunctions.
         getValueFromHeader("UserName")))
6  then
8    modify ($fo) {
9    setAttribute("balance",(Integer)($fo.getAttribute("balance")) -
         PARAM_ServicePrice);
     }
10 end
```

Listing 3. Prepaid charging rules

In Listing 3, the "check user balance" rule checks if the service requestor has sufficient funds in his/her account to access the service. The condition at Line 3 checks whether a request has been placed. Line 4 iteratively stores FactObjects which are basically representations of the registered users fetched from an external data source through the *UserMediator* that was implemented for the authentication package (as mentioned in step 1). The condition at Line 5 establishes who the user is. The condition at Line 6 checks the user's balance. If a user object is found and its balance is less than that defined through the configuration, a fault is generated and is sent back to the client. The other rule "deduct amount from service invocation" executes only for the response messages and deducts an amount equivalent to the service price from user's balance.

Scenario III:

The steps below demonstrate how the ROAD platform can be used to incorporate dynamic changes to the system functionality when additional discount pricing rules were needed to be pushed to the server without putting the server down.

When TradeAnalytics decides to place off-peak discount on the current service, the existing prepaid charging rule *deduct amount from service invocation* can be updated to include the off-peak logic as shown in Listing 4 (Line 6 and Line 9). New placeholders are introduced to configure the off-peak time and the discount percentage.

```
1  rule "deduct amount for service invocation"
2  when
3    $event : MessageReceivedEvent(operationName == "PARAM_FuntionalRequest",
                 syncResponse == true)
4    $fo : FactObject()
5    eval($fo.getAttributes("name").equals(ServiceDeliveryFunctions.
         getValueFromHeader("UserName")))
6    eval(ServiceDeliveryFunctions.currentHourOfDay() >= PARAM_CloseHour &&
          ServiceDeliveryFunctions.currentHourOfDay() <= PARAM_OpenHour)
7  then
8    modify ($fo) {
9    setAttribute("balance",(Integer)($fo.getAttribute("balance")) -
         PARAM_ServicePrice * (100 - PARAM_DiscountPercentage) / 100 );
     }
10 end
```

Listing 4. Addition of off-peak discount to the charge rule

The ROAD runtime supports a management interface through which the rules, roles and other entities in a deployed service composite can be modified. Unforeseen adaptations can be applied dynamically through this interface. Organizer utility functions are available that can be used to remove the previous rule (deleteRule function) and then to add a new rule (addRule function). The addRule function takes in the rule definition as a String. As long as the new rules do not contain new placeholders, existing rules can be dynamically updated without requiring the composite to stop and undeploy.

6. Related Work

There are different stream of work that relate to service packaging. Firstly, we present work in pattern componentization that aims to realize design patterns as reusable components. Secondly, we highlight industry approaches for adding on auxiliary service delivery functions, and thirdly, we present some other research that is closely related to our work.

6.1 Pattern Componentization

Research has been undertaken to overcome the challenge of providing reusable implementation of design patterns (Arnout, 2004), (Meyer 2004), (Hannemann and Kiczales, 2002), (Budinsky et al., 1996) and (Sorensen et al., 2010). Arnout (2004) and Meyer (2004) proposed the idea of "componentization of design patterns" i.e. turning design patterns, whenever possible, into reusable components programmed in Eiffel with a well-defined API. Hannemann and Kiczales (2002) explored how to take advantage of aspect-oriented programming to implement aspects. Budinsky et al (1996) has described a tool for automatically generating C++ code for design patterns. Sorensen et al (2010) have proposed a mechanism for writing reusable module where each module is considered as a template consisting of a collection of classes. These templates are instantiated at compile time and the ordinary classes generated can be adjusted according to the domain requirements.

The objectives of these works are similar to ours, i.e. to avoid reinventing the wheel in software construction by providing reusable implementations of design patterns that can be used by the developer's out-of-the-box. The main difference however with these works is in the level of abstraction. While we focus on high level service related patterns implementation, these works relate to low level object-oriented patterns and their application in software development.

6.2 Mediator Platforms

Cloud-based mediator platforms such as Force.com (Weissman and Bobrowski, 2009) from SalesForce and IronCloud (StrikeIron), (Mashery) and API management tools (such as 3scale) have been providing mediator functions to service providers. However, these intermediary platforms suffer from a rigid service delivery model (Weber et al., 2009) while performing the service delivery functions because of the static feature configurations offered by the platform. Considering the diversity of services offered presently, it is quite likely that there are some service delivery functions required by a service provider which are not available through a single platform, or that the business constraints does not fit with the offerings of these mediator platforms. The other limitation is that these solutions are generally heavy-weight, are focused on niche market segments and typically pivoted around outsourced hosting. Furthermore, service providers lose control over the service delivery chain if they delegate these tasks to an external entity (Baross and Dumas, 2006). Unless that external entity provides flexible ways of reshaping the service offerings, innovation in service offerings is significantly delayed or in certain cases not implementable at all.

If service providers choose to self-host the service delivery technology, middleware technologies such as an ESB or packaged applications such as billing / security frameworks are available that can be used to add-on the auxiliary service delivery functions. Scope of this research limits us from comparing our approach with the overwhelming number of industry solutions, yet we studied an industry-grade ESB (from WSO2) and a BPEL-based service composition approach. For example, the WSO2 ESB offers an easy-to-use GUIs to implement service mediation functions such as drop a message; manipulate SOAP headers, a DB lookup, etc. These functions can be applied to an incoming message amidst it being passed to the backend core service or to the response message before it is handed back to the client. Also, backend services can be encapsulated as service proxies that act as a gateway between the client and the backend services. Built-in security and logging patterns can be applied to these proxy services with a few clicks. In this regard, the ESB offers a nice solution for applying a few service delivery patterns. However, in comparison to our solution, there is no facility for the user to implement new patterns. The developer is restricted to the number of pre-defined available patterns. Another difference is that the ESB mediation follows an imperative control-flow mechanism to define the business logic whereas we use declarative rules.

A business process engine can also be used to realize reusable service packages. The process definition can be saved as a template with placeholders for the subscription or core service's endpoint addresses and/or the input/output messages. For example the BPEL abstract process (Web Services Business Process Execution Language Version 2.0) that uses abstract activities (opaque tokens or omission). The instantiation of such abstract process requires the same abstract process definition to be updated by filling in concrete details. In our approach, the abstract composition – the package; is not updated at instantiation but is merged with a concrete composition leaving the original package as-is. This separation of composite definitions allows the same package to be reused with any core composite. In addition, as compared with the procedural process-oriented approach, declarative approaches – the approach that we follow; provides more flexibility in terms of adaptation (Fahland, 2009).

Commercial Off-the-shelf (COTS) software packages for billing and security are also available e.g. (Enterprise-jBilling) that provide value added features but generally are heavy-weight. We facilitate their integration with our platform through the concept of mediators that can talk to any external system to fetch contextual information. For example, assuming a COTS billing system is being used by the service provider with a built-in rating engine that computes the service price based on a user's membership status, we can invoke this service through our platform using a mediator.

Schulz-Hofen (2007) proposes a broker based middleware that has charging, penalizing, monitoring and fulfillment services. The authors call these facilities in order to differentiate them with core services. A closely relevant facility is the charging and penalizing facility that has direct read access to the core service's pricing information and is thus able to charge a requestor directly after successful service invocation. However, no detail is provided on how the pricing information is configured or accessed. In our platform we use the configuration parameters to define core service pricing which can be configured to each individual core service and are accessed by the generic billing package.

6.3 Other Works

One of the goals of our approach is to enable non-expert service providers to configure and deploy services. In this regard, Bormann et al. (2008) share with us similar objectives. The authors have proposed an extension to the Parley X API that covers context information such as user location, userID and password, the communication protocol, connection bandwidth, etc. Their main focus is to facilitate developers developing NGN applications for the telecommunications industry. This work in based on Parlay X API standards for payment and account management and hence is focused on context aware mobile services. Another similar work targeting telecom sector is by Koutsopoulou et al (2007). We have taken a broader view of service delivery patterns instead of limiting ourselves to a specific application domain e.g. telecom or a specific set of patterns like billing.

Chang et al. (2017) proposes a methodology that allows for automatically adjusting the logical order of services on offer. Their focus is more on ordering the services based on various parameters. Jiang et al (2009) proposed a pattern based approach to design subscription management systems for SaaS applications. The paper describes the potential structure and business interaction model (the solution) applicable for a certain situation (the problem). The situation could be described in terms of the service offerings in a domain, corresponding service elements in the service offering, and their dependencies. Given a situation, this approach would then propose the structure (single service, multiple dependent services, proxy service) and the business interaction model (self-service, hub spoke, distributed, delegated) for the subscription system. This work can complement our approach by providing the preliminary design frame on top of which the developers can then start the implementation of service packages.

Sirin et al. (2005) uses Web service templates that contain abstract activities associated with preferences. A ranking

algorithm then finds the best match concrete activities based on the preferences. Focus is on finding concrete services that can fill up the abstract services. Template instantiation does not enhance the structure or behavior of the composition, just replaces the abstract activities.

Geebelen et al. (2008) describe a template based approach to dynamically reconfigure web service composition defined in BPEL. The concept is inspired by Model-View-Controller style architecture. It all begins with a standard master process which could be used to model functionality independent of concrete implementations. A library template containing the modules of BPEL activities is offered which fulfill the generic functionalities as defined in the master process. The template and the master process together are the main components to make the controller to compose a customized BPEL process. The controller then selects a suitable template and interprets numerous parameters associated with runtime environment for the BPEL process. The templates and the master process have to be written manually. Graphical design tools that can assist the designer/develop in creating these templates and the master process are future works. ROAD on the other hand provides easy to use graphical editors to create service packages, instantiate them and deployment them in a web container. We also have presented a case study on our platform to evaluate its usability.

7. Conclusions and Future Work

In this paper we have introduced the concept of a service package which is a reusable implementation of a service delivery pattern. For the package to be built and applied to core business services, we introduced a step-wise service packaging process. We explained what artefacts are produced during the packaging process. The output of the packaging process is a deployable concrete service composition model that is dynamically adaptable.

We have demonstrated that a service package is independent of a specific business domain and once implemented, can be applied to any concrete service composite that includes core business functional services. With the help of a case study, we have validated the potential of our proposed approach. The methodology along with the platform that we have presented provides a mechanism that allows developers to incrementally create and apply suitable service delivery patterns in a given situation. The approach does not necessarily require the service packages to be built ground up all the time. Once created, the pattern implementation can be reused out of the box for other services as well hence forming a library of reusable services packages.

We have leveraged the ROAD platform as the underlying technology to implement service packages. ROAD delivers the key design artefacts needed for a good service delivery solution and hence was selected as the implementation platform. The main characteristic that stands ROAD out from other middleware platform is its lightweight nature. ROAD provides a service-specific solution rather than an enterprise-wide service-bus solution across a whole portfolio of services. This characteristic of ROAD made it a good fit to model service delivery patterns as lightweight, modular, encapsulated and easy to manage artefacts. The open source tools that we have used provide a cheap alternative to the expensive middleware technologies.

The proposed framework can be used by service providers or brokers alike. If used in a brokerage environment we need to have a way to describe the service package e.g. in our Basic Authentication package, the assumption was that incoming request is SOAP and the user credentials are in the header with tags <UserName> and <Password>. If these details are not explicitly stated or exposed, it would be difficult for the developer to search for, understand, and subsequently apply an existing package. As part of the future work, we would like to explore the WS-Policies framework to expose the service package capabilities in WSDL interfaces.

Once the package is applied to a core service composite, the features of a package are merged in the core composite. If a package was to be removed from a core composite, the developer will have to manually remove the relevant elements of the package or discard the entire packaged composite and build ground-up again. This process could be laborious if done manually. Instead, an automated removal routine can prove to be handy. This may be achieved by associating a script with the package that defines the steps in which the elements/component of a package should be added or removed from a concrete composite.

Acknowledgments

Special thanks to Prof. Jun Han and Alan Colman of Swinburne University of Technology, Melbourne for their collaboration and granting me the privilege to use the ROAD platform.

References

3Scale API Management Platform. Retrieved May 29, 2018, from http://www.3scale.net/

Arnout, K. (2004). From Patterns to Components. Swiss Federal Institute of Technology, PhD thesis.

Bali, M. (2009). *Drools Jboss Rules 5.0 Developer's Guide.* Packt Publishers.

Barros, A. P., & Dumas, M. (2006). The Rise of Web Service Ecosystems. *IT Professional, 8*, 31-37.

Bormann, F. C., Flake, S., Tacken, J., & Zoth, C. (2008). Third-Party-Initiated Context-Aware Real-Time Charging and Billing on an Open SOA Platform. 22nd International Conference on Advanced Information Networking and Applications Worshops.

Budinsky, F. J., Finnie, M. A., Vlissides, J. M., & Yu, P. S. (1996). Automatic code generation from design patterns. *IBM Syst. J., 35*, 151-171. https://doi.org/10.1147/sj.352.0151

Chang, S. S., Meliksetian, D. S., & Ye, P. (2017). Apparatus, system, and method for logically packaging and delivering a service offering. US Patent US9626632 B2.

Colman, A. (2006). Role Oriented Adaptive Design. PhD thesis. Faculty of Information and Communication Technologies, Swinburne University of Technology. Retrieved from http://hdl.handle.net/1959.3/38729

Enterprise-jBilling. (2018). Retrieved March 4, 2018, from http://www.jbilling.com/

Fahland, D., Lübke, D., Mendling, J., Reijers, H., Weber, B., Weidlich, M., & Zugal, S. (2009). Declarative versus Imperative Process Modeling Languages: The Issue of Understandability. Enterprise, Business-Process and Information Systems Modeling, 29, Springer Berlin Heidelberg; pp. 353-366.

Geebelen, K., Michiels, S., & Joosen, W. (2008). Dynamic reconfiguration using template based web service composition. Proceedings of the 3rd workshop on Middleware for service oriented computing, pp. 49-54.

Hannemann, J., & Kiczales, G. (2002). Design pattern implementation in Java and aspect. *SIGPLAN Not., 37*, 161-173. https://doi.org/10.1145/583854.582436

Jiang, Z., Sun, W., Tang, K., Snowdon, J. L., & Zhang, X. (2009). A Pattern-Based Design Approach for Subscription Management of Software as a Service. Proceedings of the 2009 Congress on Services – I, pp. 678-685.

King, J., & Colman, A. A (2009). Multi-Faceted Management Interface for Web Services. Software Engineering Conference, ASWEC '09. Australian, pp. 191-199.

Koutsopoulou, M., Kaloxylos, A., Alonistioti, A., & Merakos, L. (2007). A platform for charging, billing and accounting in future mobile networks. *Computer Communications, 30*, 516-526. https://doi.org/10.1016/j.comcom.2005.11.022

Li, Q., Liu, A., Liu, H., Lin, B., Huang, L., & Gu, N. (2009). Web services provision: solutions, challenges and opportunities (invited paper). Proceedings of the 3rd International Conference on Ubiquitous Information Management and Communication, pp. 80-87.

Meyer, B. (2004). The Power of Abstraction, Reuse, and Simplicity: An Object-Oriented Library for Event-Driven Design. Owe O, Krogdahl S, Lyche T, eds. From Object-Orientation to Formal Methods, 2635, Springer Berlin / Heidelberg; pp. 236-271.

Pham, C. L., Han, J., & Schneider, J. G. (2007). Towards Dynamic Matching of Business-Level Protocols in Adaptive Service Compositions. Lecture notes in computer science, 4928, pp. 502-507.

Schulz-Hofen, J. (2007). Web Service Middleware - An Infrastructure For Near Future Real Life Web Service Ecosystems. Proceedings of the IEEE International Conference on Service-Oriented Computing and Applications, pp. 261-270.

Sirin, E., Parsia, B., & Hendler, J. (2005). Template-based composition of semantic web services. AAAI Fall Symposium on Agents and the Semantic Web Virginia, USA.

Sørensen, F., Axelsen, E. W., & Krogdahl, S. (2010). Reuse and combination with package templates. Proceedings of the 4th Workshop on MechAnisms for SPEcialization, Generalization and inHerItance (MASPEGHI '10). ACM, New York, NY, USA, Article 3.

StrikeIron. IronCloudTM. Retrieved May 29, 2018, from http://www.strikeiron.com/about-the-ironcloud-platform/

Thompson, J., Yefim, V. N., Pezzini, M., Sholler, D., Altman, R., & Iijima, K. (2013). Magic Quadrant for On-Premises Application Integration Suites. Gartner Report, ID: G00248983.

TIBCO Cloud Mashery API Management Platform. Retrieved May 29, 2018, from http://www.mashery.com/ (Accessed 29 May 2018).

Web Services Business Process Execution Language Version 2.0 (2007). Retrieved May 29, 2018, from

http://docs.oasis-open.org/wsbpel/2.0/Primer/wsbpel-v2.0-Primer.html

Weber, A. B., May, N., Hoffmann, J., & Kaczmarek, T. (2009). Composing Services for Third-party Service Delivery. Proceedings of the IEEE Int Conf on Web Services, ICWS'09.

Weissman, C. D., & Bobrowski, S. (2009). The design of the force.com multitenant internet application development platform. Proceedings of the 2009 ACM SIGMOD International Conference on Management of data (SIGMOD '09)", Carsten Binnig and Benoit Dageville (Eds.). ACM, New York, NY, USA, pp. 889-896.

WSO2. ESB. Retrieved March 4, 2018, from http://wso2.com/products/enterprise-service-bus/

Appendix A

Authentication Package service composite description:

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:SC xmlns:tns="http://www.swin.edu.au/ict/road/smc" name="AuthenticationPkg"
      isConcrete="false">
  <Roles>
    <Role id="client" name="abstract_client">
      <Description>the abstract client role that will correspond to the concrete client role</Description>
    </Role>
     <Role id="service" name="abstract_service">
      <Description>the abstract service role that will correspond to the concrete service role</Description>
    </Role>
  </Roles>
  <Contracts>
    <Contract id="cl-svc" name="abstract_client-to-svc">
      <RuleFiles>
         <RuleFile>authenticationrules.drl</RuleFile>
      </RuleFiles>
      <RoleAID>client</RoleAID>
      <RoleBID>service</RoleBID>
      <Description>abstract contract b/w client and service abstract roles</Description>
      <Abstract>true</Abstract>
      <Mediators>
            <Mediator>UserMediator</Mediator>
      </Mediator>
    </Contract>
  </Contracts>
 <Mediators>
    <Mediator id=" UserMediator" impl="PARAM_UserMediator" name="user_mediator"
pullInterval="20000">
      <Fact name="User">
        <Attributes>
          <Attribute>name</Attribute>
          <Attribute>pwd</Attribute>
          <Attribute>balance</Attribute>
```

```
            </Attributes>
          </Fact>
          <Description>this class is responsible to fetch the users from the CRM system</Description>
        </Mediator>
      </Mediators>
</tns:SC>
```

Appendix B

Concrete StockAdvice service composite description:

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:SC xmlns:tns="http://www.swin.edu.au/ict/road/smc" name="StockAdvice"
      isConcrete="true">
   <Roles>
      <Role id="customer" name="client">
         <Description>the concrete client role</Description>
      </Role>
       <Role id="service" name="svc">
         <Description>the concrete service role representing the actual service</Description>
      </Role>
   </Roles>
   <Contracts>
      <Contract id="cl-svc" name="client-to-svc" >
         <RuleFiles>
            <RuleFile>StockAdvice_client-svc.drl</RuleFile>
         </RuleFiles>
         <RoleAID>customer</RoleAID>
         <RoleBID>service</RoleBID>
         <Description>contract b/w client and svc roles</Description>
         <Abstract>false</Abstract>
         <Terms>
            <Term id="t1" name="request" messageType="push">
               <Direction>AtoB</Direction>
               <Operation name="getAdvice">
                  <Parameters>
                        <Parameter>
                              <Name>StockSymbol</Name>
                              <Type>String</Type>
                        </Parameter>
                  </Parameters>
                  <Return>String</Return>
               </Operation>
            </Term>
         </Terms>
      </Contract>
```

```
  </Contracts>
  <PlayerBindings>
    <PlayerBinding id="svcpb" name="service-playerbinding">
      <Endpoint>http://localhost:8080/axis2/services/BackEndService2</Endpoint>
      <Roles>
        <RoleID>service</RoleID>
      </Roles>
    </PlayerBinding>
  </PlayerBindings>
</tns:SC>
```

Appendix C

StockAdvice service composite description packaged with the Authentication Package:

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:SC xmlns:tns="http://www.swin.edu.au/ict/road/smc" name="StockAdvice"
    isConcrete="true">
  <Roles>
    <Role id="customer" name="client">
      <Description>the concrete client role</Description>
    </Role>
    <Role id="service" name="svc">
      <Description>the concrete service role representing the actual service</Description>
    </Role>
  </Roles>
  <Contracts>
    <Contract id="cl-svc" name="client-to-svc" >
      <RuleFiles>
        <RuleFile>StockAdvice_client-svc.drl</RuleFile>
        <RuleFile>authenticationrules.drl</RuleFile>
      </RuleFiles>
      <RoleAID>customer</RoleAID>
      <RoleBID>service</RoleBID>
      <Description>contract b/w client and svc roles</Description>
      <Abstract>false</Abstract>
      <Terms>
        <Term id="t1" name="request" messageType="push">
          <Direction>AtoB</Direction>
          <Operation name="getAdvice">
            <Parameters>
              <Parameter>
                <Name>StockSymbol</Name>
                <Type>String</Type>
              </Parameter>
            </Parameters>
```

```
                <Return>String</Return>
              </Operation>
            </Term>
          </Terms>
          <Mediators>
            <Mediator>Mediator1</Mediator>
          </Mediators>
        </Contract>
    </Contracts>
    <PlayerBindings>
        <PlayerBinding id="svcpb" name="service-playerbinding">
          <Endpoint>http://localhost:8080/axis2/services/BackEndService2</Endpoint>
          <Roles>
            <RoleID>service</RoleID>
          </Roles>
        </PlayerBinding>
    </PlayerBindings>

<Mediators>
      <Mediator id=" UserMediator" impl="PARAM_UserMediator" name="user_mediator"
pullInterval="20000">
          <Fact name="User">
            <Attributes>
              <Attribute>name</Attribute>
              <Attribute>pwd</Attribute>
              <Attribute>balance</Attribute>
            </Attributes>
          </Fact>
          <Description>this class is responsible to fetch the users from the CRM system</Description>
        </Mediator>
    </Mediators>
</tns:SC>
```