A Service Packaging Platform for Delivering Services

Muhammad Adeel Talib, Alan Colman, Jun Han, Justin King and Malinda Kapuruge
Faculty of Information and Communication Technology, Swinburne University of Technology, Victoria, Australia
{mtalib; acolman; jhan; justinking; mkapuruge}@swin.edu.au

Abstract—For a service to be used in a commercial setting, certain service delivery functions such as monitoring, billing, authentication, etc need to be packaged with the service. Because of the growing number of available services and the diversity in their offerings, predefined service delivery functions like that offered by existing service brokers (e.g. StrikeIron or Salesforce) cannot be easily adapted and plugged in with the core service from a third party provider. The need is to have a lightweight service packaging platform that offers service providers the ability to create reusable service packages by assembling existing service delivery functions or creating new ones. In this paper we introduce the notion of service packages and a pattern-based approach to service packaging. We highlight what is required from a platform to have such service packaging capabilities, and detail a prototype platform that we have developed to meet these requirements.

Keywords—service packaging platform, service delivery functions, service contracts

I. INTRODUCTION

Traditional boundaries of service supply and demand are changing as Service Oriented Architecture (SOA) efforts mature and new-generation value-added Web services emerge on the Internet of Services [1]. As organizations recognize that their services can lead to new revenue streams, they are continuously looking for ways to make legacy and new services available to different markets. For example, credit worthiness, money laundering checks, land information look-ups, address validation, trade compliance, electronic medical records, and life event registration (e.g. births, deaths, etc.) are just a few services that are being exposed by service providers such as government regulatory bodies, supply chains and health management, and so on. But just by exposing existing back-end systems through Web service wrappers does not make the service ready to be consumed in a commercial context.

There are certain ancillary service delivery functions such as metering, monitoring, billing, payment, authentication, encryption, tracking, logging, etc. that need to be packaged with the core service (the basic service offered by a provider) before it can be effectively or profitably delivered to the end consumer. Service providers newly entering the e-service arena may not have the capacity or infrastructure to provide these ancillary functions, or they simply do not want to shift their focus from their core competencies.

Traditionally, in such cases service providers delegate the provision of these ancillary services to third party service brokers or marketplaces that specialize in such functions, such as eBay, Salesforce, StrikeIron, GrandCentral, Amazon, Oracle, etc. The service provider has to register their services with these marketplaces and pay a fee. However, while these companies provide service delivery functions like payment, billing, customer account management, auctioning, etc via their brokering platforms, they do suffer from limitations of a rigid service delivery model [2] where the provider has to abide by their static service delivery offerings. This might not be a problem in a monotonic services world, but is challenging when we take in account the diversity found in today’s services.

From a simple informational service such as weather forecast to a full-scale cloud infrastructure service such as Amazon EC2, there is a huge diversity in service offerings. It is quite possible that the service delivery functions required by a service provider may not be available through the existing marketplaces. The need is to have a flexible Service Packaging Platform by which service providers can easily add these service delivery functions to their core services, can have full control over the service delivery channel and can enjoy the liberty to evolve their service offerings without incurring potentially large integration costs.

In this paper we present a platform that allows creating service delivery functions declaratively and packaging them with the core service such that the packaged service represents a reusable packaging pattern to serve a specific market context. In contrast to the existing heavy weight solutions for service delivery, our lightweight approach is provider-centric and can be used by the provider to quickly market services without incurring huge costs. The rest of the paper is organized as follows: After introducing a motivating scenario in Section II, we elaborate on the concepts of service delivery functions, service packages and patterns in Section III. Section IV highlights the requirements that have to be fulfilled by a service packaging platform. Section V discusses the approach we have taken to realize a service package and Section VI provides our platform implementation details. Related work is presented in Section VII and conclusions in Section VIII.

II. MOTIVATING SCENARIO

The service owner, TradeAnalytic, provides real-time stock exchange information for selected stocks on the Australian Stock Exchange (ASX) by providing buy/sell/hold recommendations based on the technical analysis provided by its proprietary programs. It wants to offer this service into the service marketplace so that it can be used by day-traders. To do so, it wraps its proprietary programs into a Web service wrapper and exposes it as a
StockAdvice Web service. After 6 months of free trial TradeAnalytic wants to put a fee on its usage. Also, the company has an intention to broaden its clientele in the future by offering a range of pricing options, such as, subscription based flat rate with periodical charge up to a maximum (possibly unlimited) number of queries, pay per query, a set number of queries purchased in advance, and so on. Peak hour and Off-peak hour variable rates are also under consideration. The company wants to use its own accounts management system but does not have any existing mechanism for monitoring service invocations and charging the clients based on the pricing options that they have chosen. It also realizes that the severe market competition will place a need to rapidly modify its pricings or introduce new price plans. The company therefore decides to use a service packaging platform that provides a easy to use modelling tool and a robust execution environment. The platform also allows integration with the company’s existing accounts management system and quick addition or modification to the company’s pricing plans.

After a while, the company’s analyst comes up with a more sophisticated real-time stock recommendation algorithm to assess stocks based on trading volumes, past market trends, and expert opinion. The company wishes to offer this more advanced premium advice service with the same pricing options as the standard advice but with different price tags. While the StockAdvice returns the advice immediately, the PremiumAdvice service takes more time to send a response as expert opinion is sought. The company intends to reuse some of the service delivery components it has developed for the StockAdvice service but realizes that some customization may be required to incorporate price and response time differences.

III. SERVICE PACKAGING

In our terms, a service package is a service composition whereby service delivery functions and associated business rules are applied to a core service. Service delivery functions are computational entities that perform certain functions such as extracting user/pwd from a message header, encrypting / decrypting messages, counting the number of service invocations, measuring the response time, logging, etc. They provide this information to relevant functional entities in the package. These functions are combined with business rules that govern the service contract between the provider and the consumer to form a service package. Such packages are self-contained composites that can be applied out of the box to any other service that has similar service delivery requirements without modifying the composite, and hence creating a reusable service packaging pattern. Fig. 1 pictorially describes the concepts of service delivery functions, patterns and a package. These concepts are elaborated more in Section V.

Layers of packaging patterns can be applied to the core service progressively to channelize the service offering to a specific market context as and when required. In Fig. 1, as an example, an authentication pattern is applied first to a core service for authenticating users (the inner-most doted box in Fig. 1), which is followed by the authorization pattern for validating user accounts (the middle doted box) and finally the billing pattern for billing clients (the outer doted box). Once undergone all these packaging steps, the result would be a service that authenticates users, checks the validity of their accounts, counts the service invocations and triggers billing. If required, other patterns like payments, exception handling, service tracking, logging, etc, can also be added at any packaging stage without affecting the existing packages.

Along the delivery chain, these packaging patterns can be considered to be snapshots of the packaged service that can be reverted back to. So for instance, in the above example, if the service provider wishes to enhance its offerings by introducing unlimited-usage monthly subscription billing, it can revert back to the second stage of the packaging (i.e. authentication), then add service delivery functions and business rules associated with authorization of subscription based clients, which are different from the pay-per-usage clients. This concept of repurposing services is elaborated in Fig. 2.

Figure 1. Concepts of a service package, pattern and the service delivery functions.

Figure 2. Repurposing services by adding new packaging patterns.
IV. REQUIREMENTS FOR A SERVICE PACKAGING PLATFORM

In light of the motivational scenario, this section discusses in detail the requirements that need to be fulfilled by a service packaging platform. We have categorized these requirements as design time and runtime requirements.

A. Design time requirements

Req. 1. CRUDing packages. Packages can be created, retrieved, updated and deleted (in short, CRUUDed). Diversity in service offerings caused by, for example, different pricing models [3], authentication methods [4], etc, raises the demand for creating new packages. The TradeAnalytic developer realizes that in order to put up a charging structure for its service's usage after the initial 6 months of free trial, there needs to be a mechanism to authenticate the client, validate its account for sufficient funds, keep record of the invocations and bill the client on, say, a monthly basis. To achieve this, the service packaging platform needs to provide an easy-to-use modeling environment where service providers can rapidly create service packages, save them, retrieve them for later updates, or delete them.

Req. 2. Assembling & customizing packages. Assume that TradeAnalytic now wants to launch the PremiumAdvice service. As the TradeAnalytic developer has already developed service packages for the StockAdvice service, he/she intends to reuse those packages StockAdvice but realizes that the authorization package will need customization because the premium accounts are managed in a different way than the standard stock user accounts. Furthermore, because the PremiumAdvice service involves human expert opinion, the response time is longer than the standard StockAdvice service. This requires some alterations to the authorization function in the authorization package and to the response time monitoring function in the billing package. The service provider thus requires the ability to assemble and customize pre-defined services packages.

Req. 3. Instantiating packaging patterns. A service package exists independent of a specific application and thus can be considered as a pattern. The service provider should be able to reuse a packaging pattern out of the box for its other services with similar service delivery requirements. Similarly, service providers may want to onsell their packages to other service providers. This may not require customization of the packages as described in Req. 2 but will require instantiation of the packaging patterns. For example, the same billing pattern of the StockAdvice service can be applied to the PremiumAdvice service by changing the concrete service binding (assuming the interface definitions are the same). The platform therefore should provide means to change the service bindings with little effort on the designer’s part.

Req. 4. Searching for packages. As more services become available, more packages can be developed and stored for later reuse. This requires a catalog of packaging patterns to be maintained for easy look up. Also existing repositories are not rich enough to describe the packaged content of the service and work towards enhancing the descriptive power of services is currently under its way [5, 6]. The platform should have the capability to publish-into/query-from these repositories.

Req. 5. Deploying packages\(^1\). A service package is a composition of service delivery functions and business rules. Like every composite service, the package should be deployable as a service that can be invoked through Web service interfaces.

B. Runtime requirements

Req. 6. Maintaining the execution order of the assembled packages. As a design time activity, the package designer will package service delivery functions with core services. The designer may decide to add all functions in one pattern or create patterns incrementally. If done incrementally, at runtime, the pattern applied at last will face the consumer first, e.g. the billing pattern in the Fig. 1. But the execution order should be the same as the design time packaging order e.g. the authentication pattern applied first at design time should execute first at runtime. The platform is thus required to maintain the execution order of the applied patterns at runtime which should be conformant with the designed assembly of packages.

Req. 7. Modifying existing packaged services at runtime. A change in the provider’s original offering may require runtime modification to the existing service packages. For example, consider that the company later decide to stop offering the PremiumAdvice service after ASX working hours, or it may decide to allow access to PremiumAdvice for one month free of charge for standard clients that have either reached 1000 StockAdvice invocations or have been standard clients for more than one year. To accommodate such dynamic changes, the platform should support runtime adaptation of the service packages.

Req. 8. Communicating with services or applications external to the platform. Service delivery functions provide context information to other functional entities (such as the billing system, CRM, etc) in the package. For example, the billing package used with the StockAdvice service has a usage meter service delivery function that provides the number of service invocations to the TradeAnalytic’s financial sub-systems to calculate the bill. The platform should provide means to push or pull messages to these functional entities. For example, the pay-per-use billing package may have to inform (push) the account management system after each successful invocation of the service if the consumer has opted to be billed after every invocation. On the other hand a frequent user may opt for monthly billing which will then require the account management system to inquire (pull) about the number of invocations to calculate the bill. The platform thus should have a mechanism to communicate with other applications external to the platform and should support different message exchange patterns such as push or pull.

\(^1\) Though deployment is generally another stage of software lifecycle, because we have categorized the requirements into design time and runtime only, we consider it as a design time activity.
**Req. 9. Intercepting messages.** A service delivery function could be triggered by the receipt of a message, for example, count the service requests, or could perform certain functions on the message, for example, the authentication function would extract user/pwd from the request message. The platform therefore needs to have a mechanism to intercept messages that are exchanged among the functional entities (or services) involved in packaging the core service and accordingly trigger the service delivery functions.

V. **THE APPROACH**

For the implementation of a service packaging platform, we leverage our previous work on Role Oriented Adaptive Design (ROAD) \[7, 8\]. ROAD provides a framework for developing and deploying self-managed service compositions. In this section, we analyse a service package in more detail and elaborate what a package is and what are the building blocks that comprise a reusable service package. We discuss how ROAD can be used to model and realize service packaging, resulting in a service packaging platform.

A. **Service Definitions**

A service package has to be built in such a way that it can be ported across different applications (Req. 3). This requires the package to offer interface level abstraction from the functional entities (Web services in our case). To achieve this, the service package maintains the service definitions as an abstract position description specifying the requirements for a functional entity to implement, and the binding to this implementation. The functional entity itself is external to the package and performs the domain function, e.g. the stock advice service, the billing service or the client application. If such a package has to be applied to a similar service from another service provider, then only the binding needs to be changed.

In ROAD, a role-player pair provides such indirection of instantiation. A role provides the position description (See Fig. 3) and can be considered as a proxy for a concrete service within the boundaries of a package. Roles in themselves perform no domain functions, rather all work is done by the functional entities, i.e. players – the concrete services. Roles store the endpoint references of the players to which the package is instantiated and at runtime invokes operations on the players or accept calls from them. Such a mechanism also allows a service package to communicate to external services or applications (Req. 8), where the external entities are modelled as players that play roles inside the package.

B. **Service Contracts**

Service delivery is about the provisioning of a given service to a ‘known consumer’. By known consumer, we mean a known set of users representing a market segment, for example a usage based consumer market that pays for the service based on the number of invocation, or unlimited monthly subscription users. Service packages are created to target specific market segments. Anonymous services that operate without requiring the identity of the consumer at runtime, for example a stock price service, do not need service packaging. In order to deliver a service to such market segments, the service providers need to understand their consumer’s demands. Consequently a service package needs to capture the business rules – the contract that binds the consumers with the services. These rules hold the terms under which a service provider agrees to provide the service and the consumer agrees to pay for it. Some examples of these contracts are, ‘if response time is greater than 10 seconds, the service provider will not charge the client’, ‘if service invocation is made after hours, access will be denied’, ‘if consumer is on a monthly unlimited plan, the service provider will send the bill at the end of monthly billing cycle’, and so on. These contractual requirements are obligations on each role in the package and have to be captured and should be available for runtime modifications (Req. 7).

In addition to the obligation rules, there are: conversation rules that govern the interaction protocol between the roles, for instance ‘if a service is invoked, the service acknowledges the request first and sends the reply later’; permission rules that define the conditions under which an interaction is permissible, for example, ‘if user is not authenticated then do not allow invocation’; and state management rules that define the life-cycle management rules of the contracts, for example, ‘if subscription is about to expire, remind the consumer’.

ROAD offers the concept of contract (the diamond shape in Fig. 3) that accords well with the above mentioned contractual rule management requirement for a service package. We have extended ROAD’s concept of contracts by formalizing the various types of contractual clauses in a declarative manner (i.e. Event-Condition-Action rules). This allows developers to write and manage rules using open source rule engines easily. Also, it allows dynamically modifying the rules at runtime.

C. **Service Delivery Functions**

For the business rules defined through contracts to fire, facts are required by the rule engine. For example, a billing pattern packaging a service offered for a pay-per-use market segment has a business rule stating that the provider will not charge for a service if the response time exceeds a certain time limit. Such a package needs to get the value of the

![Figure 3. Components of a service package.](image-url)
response time to take necessary actions. To get the value, a service delivery function ‘response time monitor’ is used by the package. Service delivery functions perform some computation and provide information to the relevant functional entity in the package. Their job is to provide context information to function entities (roles). Multiple service delivery functions can be added to a package. Other examples of service delivery functions are: a usage meter that counts the number of invocations and provides the count to the account management system for record; a user authentication function that extracts user information from the message header and passes it on to the customer relationship management system to authenticate users; a logging function that captures a fault event and pass it on to the logging system to log the event based on the fault severity.

In contrast to a player which is a functional entity external to the service packaging platform (even though it could be within the organizational boundaries of the service provider), service delivery functions are provided by the service packaging platform and work on the messages being exchanged between the service and the consumer (Req. 9). Fig. 4 shows an example of a service package with user authentication, usage meter and event logger service delivery functions applied to the Client-StockService contract and providing context information to the CRM, Accounts Management System and the Logging System respectively.

We added this feature in our implementation by defining the service delivery function in the form of Event-Condition-Action (ECA) rules. These rules can be applied to any contract in the package. The runtime will be listening to the messages being passed between the functional entities. The condition part of the rule determines the party from which the message is arriving and uses temporal operators to relate the messages. The action part defines the logic of the function to be performed by the service delivery function, e.g. increment the counter, push the counter value to the financial entity, extract user information from message header, etc.

D. The Package

A package is essentially a combination of service definitions linked to concrete functional entities, contracts between the functional entities, and the service delivery functions that provide context information to the functional entities. A package is a self contained and modular composite that is instantiated by concrete functional players. It creates a service delivery system for the underlying core services, conceptualized as a reusable pattern that can be assembled with other packaging patterns to meet diversified business goals.

The Self Managed Composite (SMC) of ROAD (i.e. a ROAD composition) is akin with the behavior of a service package, and is used to model a service package (Req. 1). The SMC captures the structure (components) and the behavior (contractual rules) at different levels of granularity (Req. 2), where the coarse-grained composites are composed of finer grained composites in a way that the latter play roles in the former. The SMCs are deployed in a Web service container and the runtime provides service invocation mechanism (Req. 8). The runtime maintains the composition order which is exactly in-line with the Req. 6 of maintaining the execution order of service packages. At runtime business rules are allowed to be modified (Req. 7). All runtime communications between the functional entities bound to the package is via the SMC that intercepts the message being exchanged between the functional entities (Req. 9) and pass these on to service delivery functions or other components as required.

VI. IMPLEMENTATION

This section presents the implementation of our prototype platform followed by the application of a use case from our TradeAnalytic scenario.

A. ROAD4SDF – A tool chain

We have implemented a tool chain (Fig. 5) which we refer to as Role Oriented Adaptive Design for Service Delivery Framework (ROAD4SDF)². Using this tool the service provider can: create service packages in a model-

![Figure 4. Interplay of service delivery functions with functional entities in a service package.](http://www.swinburne.edu.au/ict/research/cs3/road/)

Driven way, save these models for later reuse and deploy them in a Web service environment. ROAD4SDF offers a wide range of runtime execution control, monitoring, and adaptation capabilities. The toolchain is in the form of a series of modular components which are built on top of open-source tools/APIs like JBoss Drools, JAXB, Axis2 and Eclipse.

The ROAD4SDF toolchain consists of three major components each designed and implemented as a separate but compatible module. When these components are used together they provide the ability to deploy fully functional service packages (or SMCs) into a Web service container. The first is the ROADdesigner, the eclipse based modelling environment (screenshot shown in Fig. 6) that provides a graphical editor to create service packages in the form of SMCs (Req. 1) that can be saved for later reuse (Req. 2). It allows structuring the composite and capturing the business rules using graphical dialogs. The designer also provides a facility to publish / enquire about service descriptions to / from a service repository (Req. 4) so that roles can be bound to concrete services for instantiation (Req. 3). Once modelled, the package definition is exported as an XML file. The rules captured through the rules wizard are automatically transformed into the Drools rule language.

The second tool component is the ROADfactory that takes the XML file and instantiates an in-memory model of the SMC using JAXB. The ROADfactory provides a window to the outside world through a specialised Role called the Organiser (the explanation of this is outside the scope of this paper, for details the reader is referred to [8]). Through this role, an entity (whether human or machine) can get access to the internal structure of a runtime composite, allowing addition / removal of business rules as well as structural reconfigurations (Req. 7) at runtime. The JBoss Drools engine (which the ROADfactory is built on) takes the rules, listens to events and triggers actions.

Lastly, the third tool component is the ROAD4WS which is an extension to the Axis2 Web service engine. The ROAD4WS deploys the runtime SMC (Req. 5) in a Web service environment where roles are exposed as services. It provides a mechanism to intercept the SOAP messages (Req. 9) and pass them on to the ROADfactory that then triggers the service delivery functions to perform their defined actions (using the Drools rule engine). Processed messages are either pushed or pulled by the players (Req. 8) through the exposed roles. The deployed SMC represents a service packaging pattern. Another pattern applied on top of this deployed SMC will refer to this SMC as a player of a role in its SMC, hence creating a hierarchy of composite service packages. The runtime maintains this hierarchy in which the patterns are applied (Req. 6).

B. Using ROAD4SDF for packaging services - A use case
With the help of the motivational scenario presented in Section II, we now elaborate how ROAD4SDF can be used to quickly develop reusable service packages. Let us consider the use case where TradeAnalytic wants to setup a billing package for its core service StockAdvice that is targeted towards monthly usage based subscribers. The service also wants to enforce response time quality compliance. In case of response time more than 10 seconds, the client is not billed. Fig. 7 shows this scenario of service packaging.

At design time, using the ROADdesigner the TradeAnalytic developer starts to create the service package. The developer creates the StockAdvice and Client roles and places the contract between the roles. The developer then adds a usage meter service delivery function to the contract and defines a rule: ‘when a response is received within 10 sec of a request, increment counter’ Fig. 8 shows the rule in Drools format. Then a response time monitor is added with the rule: ‘when a response is not received within 10 sec of a request, measure the time difference and send to the billing system’. These rules are captured through a rule wizard (in the ROADdesigner) and automatically transformed into Drools rule format.

TradeAnalytic has a billing sub-system consisting of three functional entities: Biller, Accounts Management and the Client Payments. These roles are bound by the subscription contracts at the financial level, so the rules captured here would be like ‘if response time is less than 10

**Figure 6. Screenshot of ROADdesigner.**

**Figure 7. TradeAnalytic billing service package.**

```
rule "UsageMeter"
when
  $req : TermExecutedEvent(msg == "StockAdvice")
  $res : TermExecutedEvent(msg == "StockAdviceResponse", correlationId == $req.correlationId,
    this after[5s, 10s] $req)
then
  counter++; // increment a global variable

rule "SubscriptionCompliance"
when
  $req : TermExecutedEvent(msg == "StockAdvice")
  $res : TermExecutedEvent(msg == "StockAdviceResponse", correlationId == $req.correlationId,
    responseTime > 10s $req)
then
  measureTimeDelta(responseTime, $req)
  sendToBillingSystem(timeDelta)
```

**Figure 8. A Drools rule for usage meter.**
seconds, then charge the client’, ‘if subscription is monthly then pull counter value at the end of monthly cycle and send the bill’, and so on. The response time and usage meter functions provide the response time for delayed invocation and the invocation count respectively to the billing sub-system through the Biller role which in turn triggers billing processes based on the client’s subscription contract. The Client Payments role provides a service interface where consumers can query for their account’s status, add credit, subscribe, or perform other account management functions.

Once modelled, the service package is deployed into the ROAD4WS container. At runtime a client application (player) invokes the StockAdvice operation on the Client interface (role). The StockAdvice request message is intercepted by the ROADfactory runtime and an event is generated with a timestamp. Assuming the user has been authenticated and the account is validated, the message is passed on to the StockService role which in turn invokes the corresponding operation on the StockAdvice service player. Once a response is received from the player, another event is generated with a timestamp. The usage meter rule (Fig. 8) then increments the counter based on the response time condition. The Biller role will then pull the counter value at the end of the billing cycle which is defined in the Biller-Accounts Management contract (for example weekly or monthly billing). This deployed package and the rules are available at runtime to the developer for dynamic modification, for instance shortening the response time compliance to 5 seconds. Fig. 9 shows screenshots of the runtime client applications. The client program (consumer of the service) invokes the service a defined number of times. For experimental purposes a delay is induced in the service at runtime by holding up the service response for more than 10 sec (by the StockAdvice program displayed as TradeAnalytic Recommender in the Fig. 9). The bill which is then generated does not include the service invocations with delayed responses.

VII. RELATED WORK

Current service marketplaces such as StrikeIron, Oracle, Amazon, Salesforce, etc offer proprietary service delivery platforms but they suffer from two key limitations: Firstly, they assume a fixed service delivery model where services are ordered, invoiced, and paid through demand-usage [2], i.e. they prescribe some limited predefined service delivery functions to choose from. Service providers do not have the liberty to evolve their service offerings. Secondly, they offer heavy weight solutions that are expensive and time consuming to use or adapt and in some cases offer only service delivery for Web applications [9]. Our platform allows more flexibility and extensibility than the current service delivery platforms. It allows service owners to create their own service delivery functions and stay in control of their delivery channels.

In the past few years, Service Delivery Platforms (SDP) [10, 11] have emerged in the telecommunication sector to manage delivery of communications, media, and entertainment services from communications service providers to mobile users. Unlike our Web services-oriented delivery platform, the telco SDP provides an infrastructure for real-time high performance telecommunication services and is dominated by proprietary and custom IP technologies. Like any SOA based solution, our approach can be applied at the application level of an SDP for integration between the SDP and Business Support Systems, where they naturally fit.

Some researchers have taken a third party service broker view of service delivery following the concept of Web service ecosystem proposed by Barros and Dumas [12]. In [2] Weber et al. take a semantic annotation approach to plug-in service delivery functions into intermediate steps of long running business processes. They model service delivery functions as business entities with lifecycles described through state machines. The service provider specifies pre-and post-states wherever required in their process. The states that represent the service delivery functions are predefined by the broker. We on the other hand model service delivery functions declaratively in the form of ECA rules where new functions can be defined and hooked into the composition. The rule engine then executes the rules that in turn govern the execution follow of the process. Schulz-Hofen [9] proposes a conceptual Web service middleware and lists some requirements for it such as the need for a registry, charging and penalizing mechanism, monitoring and fulfillment functions, QoS statistics, and authentication. Though we also advocate the need for these service delivery functions, our approach is not confined to these. We encourage creation of new reusable packaging patterns using these and other functions. Furthermore, they did not provide an implementation for such a middleware. Although in this paper we have taken a provider oriented view of a service delivery platform, our platform can also be used by service brokers to deliver third party services. This is achievable because of the indirection of the abstract service definition (role) from the concrete service (player) that is supported by our platform, but may raise additional challenges regarding trust and shared cost of service.

There is a huge body of work on Web service composition platforms (e.g. [13-15] just to name a few). Generally, their focus is on automating dynamic service composition based on some defined goals and quality constraints, where the primary objective is to compose functionally compatible core services. A service package
developed through our approach is also a composite that composes different functional entities in a package. The difference is that instead of finding services that match the application requirements our emphasis is on adjoining service delivery functions with "a" core service so that the core service becomes marketable. A service delivery function itself is not a service but a piece of functionality. We also give more emphasis to the market context in which the core service is used, i.e. the declaratively captured contractual agreement between the service provider and the consumer.

VIII. CONCLUSIONS

In this paper we have introduced the notion of service packaging for delivering services to a consumer in a commercial setting. A service package is a generic design and deployment template supporting a specific market context which can be repurposed to accommodate growing market demands and business contexts. The package utilizes service delivery functions such as authentication, metering, billing, security, etc as well as the business rules that govern the contractual agreement between the service provider, the consumer and any other functional entity involved in the package. Packages are self-managed modules that can evolve without affecting other packages included in the service delivery chain.

We have extended the ROAD framework to develop a lightweight service packaging platform, where a service package is modeled and deployed as an adaptive service composition. Service providers can use the platform to quickly add ancillary service delivery functions to their core services and push them into the marketplace. The platform provides an end-to-end service delivery solution from modeling to execution, is based on model-driven architecture, offers runtime adaptation capabilities, and is implemented in the form of modular software components that are extendable.

As contractual clauses and the service delivery functions are captured as rules, our approach involves extensive rule engineering effort which we would like to automate in the future. This can be achieved by transforming service policies, SLA, etc into rules by defining the transformation mappings. Also, currently runtime modifications to the package are done by directly updating the rules and the package description file(s). We need a runtime management interface that facilitates the designer in applying runtime changes. Another on-going effort is to distill some typical service packaging patterns for general use in the service delivery context.

ACKNOWLEDGEMENTS

This work is partly supported by the Australia’s Smart Services CRC through its Service Delivery Framework (SDF) project. We thank Swapnil Raverkar and Vishnuvaradhan Jayakumar for their contribution towards the development of the ROAD4SDF tool chain. We also thank our SDF collaborators for their useful feedback.

REFERENCES