ROAD4SaaS: Scalable Business Service-Based SaaS Applications

Malinda Kapuruge, Jun Han, Alan Colman, and Indika Kumara

Faculty of Information and Communication Technologies
Swinburne University of Technology, Melbourne, Australia
{mkapuruge,jhan,acolman,iweerasinghadewage}@swin.edu.au

Abstract. Software-as-a-Service (SaaS) is a software delivery model gaining popularity. Service Oriented Architecture (SOA) is widely used to construct SaaS applications due to the complementary characteristics in the two paradigms. *Scalability* has always been one of the major requirements in designing SaaS applications to meet the fluctuating demand. However, constructing SaaS applications using third-party *business services* raises additional challenges for the scalability of the application due to the partner services' variability and autonomy. Any approach used to develop scalable service-based SaaS applications that compose business services needs to consider these characteristics. In this paper we present an approach to deploy *scalable business service compositions* based on the concept of an extensible hierarchy of virtual organisations. The explicit representation of relationships in the organisation allows capturing commonalities and variations of relationships between business services while its extensibility allows scale-out/in the SaaS application instance.

Keywords: SOA, SaaS, Scalability, Service Variability.

1 Introduction

Software-as-a-Service (SaaS) is a software delivery model that allows software users (SaaS tenants) to use the software provided by a software vendor (SaaS vendor) on a pay-as-you-go basis over the Internet [1, 2]. The SaaS vendor owns and maintains the software system and its infrastructure, whilst the SaaS tenant pays a subscription fee to use the software system. The SaaS vendor exploits the economies-of-scale available from sharing resources and services between multiple tenants, whilst the SaaS tenant benefits from low start-up-cost and quick return-on-investment [1, 3].

SaaS is not a software construction model but a software delivery model [2]. Service Oriented Architecture (SOA) provides a suitable software construction model for SaaS. As such, a SaaS application can be exposed as a service and delivered to a variety of tenants. In addition, a SaaS vendor can outsource certain functionalities of its SaaS application to third party services (partner services) and can bind/unbind them depending on fluctuating demand, making it a dynamic service composition.

The fluctuating demand may be practically impossible to predict at the system design time. Contracting and binding a large number of services in the composition,

C. Salinesi, M.C. Norrie, and O. Pastor (Eds.): CAiSE 2013, LNCS 7908, pp. 338–352, 2013.
© Springer-Verlag Berlin Heidelberg 2013

may give the SaaS vendor the capability to deal with the increasing demand but may not be an economical solution when the demand is low given expenses associated with keeping them contracted. The cost-per-unit can increase, making tenants look for alternatives. On the other hand, failure to meet the increased demand may potentially damage the SaaS vendor's business reputation. Hence, a SaaS vendor has to strategically *scale-out* or *scale-in* its service composition depending on the demand. The scalability of the system plays an important role in achieving this objective.

The *scalability* is a desirable property of a system, which indicates its ability to handle growing amount of work in a graceful manner [4]. As such it should be possible to cater for the increased demand with minimal interruptions to ongoing operations of the system. There is a substantial amount of work addressing such issues in terms of data and computational resources. For example, multiple data storages [4] or computational service/server instances [5] are bound and released depending on the demand. However, such solutions fall short when applied to *SaaS applications that compose business services* for two main reasons. Firstly, the *business services* are not homogenous. As such, it is not practical to assume all the available business services to perform outsourced functionality are alike. Unlike storage or computational service instances, there is variability even between *functionally similar business services* and consequently between the *business relationships* among the partner services in a SaaS application. Such variability needs to be captured in the SaaS application design. Secondly, business services are autonomous and managed by third party business organisations. The business relationships between its partner service providers may change over time. The up-to-date business relationships need to be explicitly reflected in the IT design. The inability to sufficiently and timely address these requirements can be problematic for a SaaS vendor.

To address the above limitations, in this paper we propose a novel methodology and middleware platform, ROAD4SaaS, to support the design and deployment of SaaS applications that compose business services. ROAD4SaaS provides a scalable and adaptable design that can be used to scale-out/in the SaaS application economically by binding/unbinding partner services to meet the fluctuations in demand while preserving the heterogeneity in service relationships. The entire SaaS application is modelled as a hierarchy of organisations. For the purposes of this approach we define an organisation as a service composition consisting of roles played by other clients/services with respect to the organisation. A structure over these roles defines and regulates the relationships between role players. The key benefit of such a design is its ability to explicitly capture the commonalities and variations of business relationships among partner services in the scalable organisation hierarchy. Sub-organisations can be created that handle and hide the complexity of particular business functions. In addition, an organisation (node) in the organisation hierarchy is adaptable to accommodate the changes in business relationships.

The rest of the paper is organised as follows. In Section 2, we analyse the problem by presenting a motivational business scenario. The approach and its prototype implementation are presented in Sections 3 and 4 respectively. The evaluation results for our approach are given in Section 5. In Section 6, we discuss the related work and provide a comparative analysis of our work before the paper concludes in Section 7.

2 Problem Analysis

In this section we analyse the problem by presenting a motivational business scenario and a set of challenges in designing SaaS applications as business service composites.

2.1 Motivation Example

RoSAS (Roadside assistance as a service) is a business organisation that expects to provide *roadside assistance as a service* on demand. Other companies such as car vendors and travel agents wish to attract customers by offering roadside assistance as a value added service but do not possess the desire, capacity or expertise to own and operate such a system on their own. These companies may use RoSAS's roadside assistance service (exposed through a software service) on *subscription* basis [3]. As SaaS tenants, they benefit from the intrinsic properties of SaaS such as lower start-up cost and quicker return-on-investment compared to creating and operating their own roadside assistance service systems.

On the other hand, RoSAS creates business value by contracting and integrating a number of third party business service providers such as Tow-Truck, Garage and Call Centre services to tow stranded cars, repair damaged cars and handle claims respectively. These third party service providers expose their offerings through software services, which we refer to as *business services*, e.g., a tow request accepting service is exposed by a Tow-Truck company. In this context, the RoSAS business model can be fittingly modelled and enacted as a service composition (IT model) following SOA principles. However, RoSAS faces a number of challenges in designing its SaaS application in terms of how the application should scale-out/in as the demand fluctuates.

2.2 Scalability Challenges for Business Service Compositions

During the runtime, the demand for roadside assistance may fluctuate. Many tenants, who themselves may have thousands of customers, are expected to subscribe to RoSAS. In addition, during peak periods, such as holiday seasons or bad weather, the demand for roadside assistance may increase compared to rest of the year. While it is convenient to assume that bound partner services, e.g., Garage chains/Tow-trucks chains are responsible to scale-out/in their operations to cater for peaks and troughs in demand, it should not be overlooked that partner services too have limitations of real-world resources [6], e.g., *number of repair stations of a Garage chain*. The failure of its partner services to meet tenants' demand risks putting the reputation of the SaaS vendor at stake [3]. In practical circumstances it could be difficult to find a single partner service, which is capable of meeting the overall increase of demand. On the other hand, contracting with a redundant number of partner services might not be an economical solution when the demand is low, making it more economical for RoSAS to contract partner services depending on the demand fluctuations during runtime.

With the increased adoption of the cloud computing paradigm, the need for such *scalable design* is well-understood [4, 5]. For example, multiple data storage [4] or computational service instances [5] are bound and released as warranted by the

demand. However, the situation is different when it comes to business services such as Tow-truck chains and Garage chains due to the following reasons.

1. In practice, business services are not homogenous as data storages or computational service instances in terms of business aspects. There are varying business requirements and relationships. For example, one garage chain might need a bonus payment for every 10^{th} repair request whilst another might be instead satisfied by an advance payment with each repair request.
2. Typically, business services are autonomous and managed by third party business organisations. The ever-changing business services and relationships may demand changes to composites that bind such services such as RoSAS. For example, the bonus payment will be paid every 5^{th} request instead of every 10^{th}. Therefore the service composites that bind business services need to be highly adaptable to continue functioning upon such changes.

These differences in composing business services raise challenges to the SaaS vendor who integrates *business services* compared to an IaaS/PaaS vendor who integrate storage or computational service instances. In the light of above differences, the solutions [3-5, 7, 8] used at the IaaS/PaaS level are not sufficient to scale-out/in SaaS applications built by composing business services. The variations in business services and relationships need to be accounted for in designing a scalable SaaS application. Therefore the design methodology used to compose business services play an important role. As such, we identify the following requirements that should be satisfied by a business service composition methodology (*Req1, Req2*) and supporting middleware (*Req2, Req3*) in order to be effective in modelling and enacting SaaS applications.

- *(Req1):* The design of a SaaS application needs to be extensible, so that number of services accommodated can be increased or decreased.
- *(Req2):* The commonalities and variations of business services and their relationships need to be clearly represented in the design and managed at runtime.
- *(Req3):* The middleware needs to ensure that adaptations to a SaaS application are carried out with minimal disruption to the ongoing operation of the composition.

3 The Approach

In this section we present our approach to achieving scalable service-based SaaS applications. After giving an overview, we describe how a SaaS composition can be designed following an organisational paradigm. Then we present how scalability and variability requirements are supported.

3.1 Overview

To address the aforementioned challenges, we design a SaaS application as a *hierarchy of organisations*, where the partner business services and their relationships are explicitly captured and represented in the organisation design. The organisation hierarchy can scale-out/in to accommodate more/less partner services (Section 3.3), while capturing the commonalities and variations (Section 3.4).

Each *organisation* (a node) in the hierarchy consists of a set of well-defined roles and relationships between them. The *roles* represent the participants and their capabilities needed by the organisation, and can be fulfilled or played by *atomic players* (i.e., both service providers and consumers) or other *organisations* of the hierarchy. The relationships are represented as contracts to capture and enforce the business relationships among two roles. A *contract* captures the allowed interactions between two roles via a set of *Interaction Terms* and its current state via a set of *Facts* (key-value pairs). A contract also defines a number of *Rules* to enforce the relationship. Both roles and contracts of an organisation are adaptable to guarantee that the organisation structure in the IT model reflects the up-to-date services and their relationships in the business model/environment. Summarising the above concepts ROAD4SaaS meta-model is presented in Fig. 1.

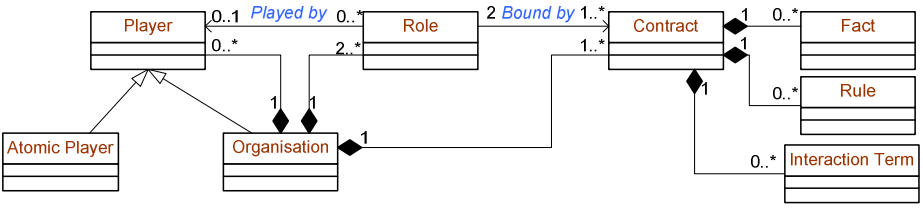


Fig. 1. ROAD4SaaS meta-model

Note that forming an organisation hierarchy is possible, because a player of a role could also be an organisation (Fig. 1). In such an organisation hierarchy there is always a *root organisation* which is also considered to be the *initial design*. In addition, there can be a number of *sub-organisations* as intermediary nodes of the hierarchy introduced to scale-out the application. The leaf nodes are always the specific atomic players, whom composition is unknown or extraneous.

3.2 The Initial Design (Root Organisation)

The initial design of SaaS composite (root organisation) provides the abstraction over the business environment. The required functionalities that need to be fulfilled by services are identified and decomposed into a set of roles. Also, the relationships among these roles are identified and represented as a set of contracts. Such organisational structure provides a *virtualisation layer* over the available concrete services.

Fig. 2, shows the root organization in the service composition for our motivating example. As shown, the root organisation captures four roles, Member (MM), Call-Centre (CC), Tow-Truck (TT) and Garage (GR), which represent the required functionalities that are expected of and outsourced to third party business services. For example, *FastRepairs*, which is a garage chain business, may bind to role GR. Once bound, the repair requests are forwarded to the provided service endpoint for *FastRepairs*. The organisation defines contracts MM-CC, CC-TT, GR-TT, CC-GR between these roles based on the requirements of supporting interactions and maintaining relationships, e.g., CC and GR need to interact and maintain their relationships, and hence

the CC-GR contract is defined in the context of RoSAS. However there are no such interactions required between MM and TT and hence no contract is defined.

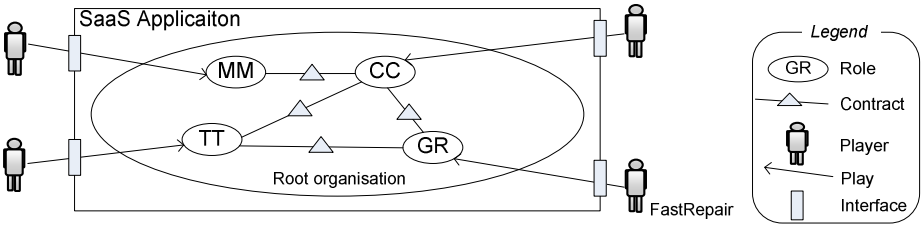


Fig. 2. The initial design

It should be noted that third party business services are autonomous and may change their behaviour during runtime. Similar issues have been identified in component-based software design and the use of contracts [9] is equally applicable in the context of composing business services too. Also, the service relationships that exist in the business model need to be explicitly represented in the composition or the IT model [10]. Therefore, to describe the objectives of SaaS vendor, we capture a contract between two roles of a composition as Interaction Terms, Facts and Rules. Here,

Interaction Terms: A set of allowed interactions between two roles.
Facts: A set of parameters that describe state of the contract.
Rules: A set of rules that evaluate the interactions of the contract.

An example contract between CC and GR is shown in **Fig. 3**. The contract has two facts, i.e., TotalRepairCount and AllowedRepairTypes which collectively represent the state of contract CC-GR. The three interaction terms (ITerm) defines all the possible interactions between the CC and GR. For example, the iOrderRepair defines the parameters (repairInfo, caseId) and directions of the interaction, i.e., from CC to GR. The rules (RuleFile) define how the interactions/messages should be evaluated against the current state of a contract. We use Drools [11] to define such business rules.

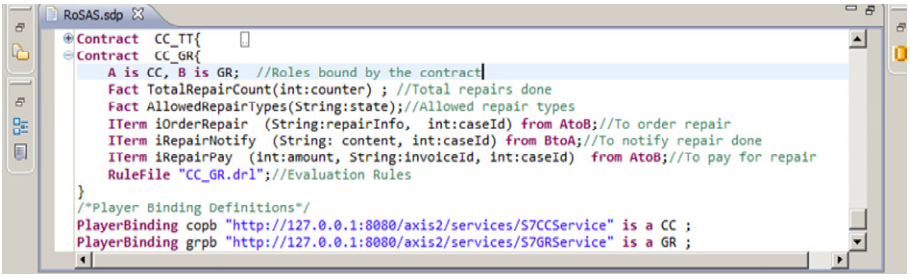


Fig. 3. A sample contract description

3.3 Supporting Scalability

The *scalability* is required to handle a growing amount of work in a graceful manner [4]. Note that there are two types of scalability, i.e., *vertical* (scale-up) and *horizontal*

(scale-out). The vertical scalability is achieved by adding more resources to a node, whereas horizontal scalability is achieved by adding more nodes [4].

This work focuses on horizontal scalability in order to resolve the bottleneck of limited partner services (nodes) from a service aggregator (SaaS vendor) perspective rather than increasing the capability of a single node, e.g., *the computing power of a computing node / repair capacity of bound Garage*, which is a separate matter of concern. Support for scalability in system design improves its *elasticity*, which primarily is a resource provisioning concern [4, 6]. In this work, the scalability of SaaS composite is supported by scaling-out or scaling-in the organisation hierarchy so that more/less partner services can be accommodated for the SaaS application. Provided that SaaS vendor has finalised the business level negotiations with suitable partner services, we explain the *scale-out* and *scale-in* operations in IT support as follows.

Scale-Out. The scale-out operation is carried out on an identified role called *expansion role* (ER) by creating a new *expansion organisation* (ER_ExpOrg). We introduce an *scale-out process*, described in Fig. 4, which scale-out a recognised *expansion role (ER)* for a given *set of players/partner services (P[]) and* for a given *Routing Strategy (S)*. The routing strategy specifies how the incoming jobs are distributed, e.g., *round-robin*, *content-based routing*. The scale-out process starts by creating a new *expansion organisation* (ER_ExpOrg) and creating a new *router role* (ER_r) provided ER_ExpOrg does not already exist. The purpose of the ER_r is to route the incoming jobs among other roles of the new organisation using the provided routing strategy S. The URL of ER_r is used as the player of ER making role ER_r the delegate of ER_ExpOrg. Then a set of functional roles (ER_i,$i \in N$) and a set of contracts between ER_r and ER_i are created to be bound by *P[]*. Here *rIndex* is the number of functional roles (ER_i) exists in ER_ExpOrg. Each created contract is populated with the *Role Interaction Description* (RID) of ER (explained below). Note that, in the case of creating new ER_ExpOrg, the currently bound player of ER could be included in *P[]* to retain in the composite.

```
function scale-out(ER, S, P[]){
  if(ER_ExpOrg does not exist){
    create new organisation ER_ExpOrg;
    create new role ERr in ER_ExpOrg;//Router role
    bind URL of ERr to ER;//Make ERr a representative of ER_ExpOrg
  } else {  retrieve ER_ExpOrg;  }
  assign new routing strategy S to ER_ExpOrg;
  for (i= ER_ExpOrg.rIndex; i<=(P.size + ER_ExpOrg.rIndex); i++){
    create new role ERi in ER_ExpOrg;
    create new contract ERr-ERi;
    populateContract(ER, ERr-ERi);
    bind P[i] to ERi;
  }
}
function populateContract(ER, C){
  Iterm[] iTerms = getRID(ER);//See Eq. (1)
  for(each Iterm it of iTerms){  C.addIterm(it);  }
}
```

Fig. 4. The scale-out process

For example, suppose the role GR needs to be expanded to assign another two new garage chain services, e.g., *BestRepairs* and *AceRepairs* in addition to existing *FastRepairs*. In this case the scale-out function is called as follows.

scale-out(GR, routing.drl, <FastRepairs, BestRepairs, AceRepairs>);

The resulting hierarchical organisation structure is shown in Fig. 5. The new composite GR_ExpOrg is created with new contracts GR_r-$GR_{[1-3]}$. As shown the existing player *FastRepairs* is now bound to role GR_1, whilst new players, i.e., *BestRepairs* and *AceRepairs* are bound to GR_2 and GR_3 respectively. The role GR of parent organisation is now bound by GR_r representing GR_ExpOrg.

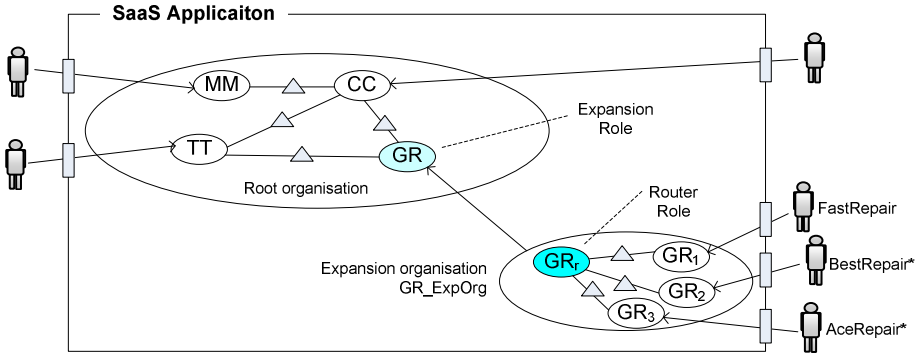


Fig. 5. Expansion organisation

Role Interaction Description (RID). The newly created contracts ER_r-ER_i need to conform to the parent composition. In this sense, messages flow across the contracts of parent organisation to a role needs to flow across its expansion organisation too. Therefore we populate new contracts with RID. For a given ER, having *m* number of contracts with *Adjoining Roles* AR_j ($j \in N$),

$$RID_{ER} = \bigcup_{j=0}^{m} \left[\bar{I}.(AR_j\text{-}ER)\right] \cdots\cdots\cdots \quad \cdots(1)$$

Here, $\bar{I}.(AR_j\text{-}ER)$ is the set of interaction defined in contract between AR_j and ER. AR_j is a role that has a contract with ER.

For the given example (Fig. 5) expansion role GR, has two *Adjoining Roles* CC and TT. Therefore, the RID of GR is *all* the interaction terms defined in CC-GR and TT-GR. Suppose that CC-GR has three interaction terms {iOrderRepair, iRepairNotify, iRepairPay} and TT-GR has two interaction terms, {iInformRepairStation, iInformDelay}then the RID of GR is,

$RID_{GR} = I.(CC - GR) \cup I.(TT - GR)$

= {iOrderRepair, iRepairNotify, iRepairPay, iInformRepairStation, iInformDelay}

Accordingly, each new contract GR_r-GR_1, GR_r-GR_2 and GR_r-GR_3 are populated with RID_{GR}. This allows the messages flow across both CC-GR and TT-GR to be routed to respective players bound to GR_ExpOrg.

It is also possible to expand a role of an expansion organisation creating another level in the hierarchy as an alternative to adding new roles. However, such scale-out

is advised only if that helps to capture commonalities and variations (explained in Section 3.4) to avoid complexity of having needlessly many levels.

Scale-in. SaaS providers may decide to remove some partner services from the composition in low-demand periods. Hence, we introduce the *scale-in process*, described in **Fig. 6**, which removes a set of players *P[]* from a given expansion organisation *ExpOrg* and updates the routing strategy with *S*. *Scale-in* is a reversing process of *scale-out* that either removes a subset of roles and their players from an ExpOrg (if the number of roles of ExpOrg, N >= P.size+2) or removes the complete ExpOrg otherwise. In the case of subset of role (N>P.size+2), a new routing strategy S is assigned to ExpOrg. In the case of removing the ExpOrg(N=P.size+2), the endpoint of only remaining player is bound to the ER of parent organisation. It is not possible to remove more players than bound (N< P.size+2).

```
function scale-in(ExpOrg, S, P[]){
  if(ExpOrg.N<=P.size+2){
    for(i=1; i<=P.size; i++){
      remove p[i].Role;
      remove contract ER_r-(p[i].Role);
    }
    assign routing strategy S to ER_ExpOrg;
  }else if(ExpOrg.N == P.size+2){
    bind remaining player to ER of parent;
    remove ExpOrg;
  }else{   //ERROR. Not Possible   }
}
```

Fig. 6. The scale-in process

Overall, the scalability of SaaS application is supported by the hierarchical service decomposition provided by the organisational approach. As shown in Fig. 7, the scale-out/processes allow growing (t0→t1→t2) and shrinking (t2→t3) the organisation hierarchy to accommodate more services when the demand is high or remove existing services when the demand is low.

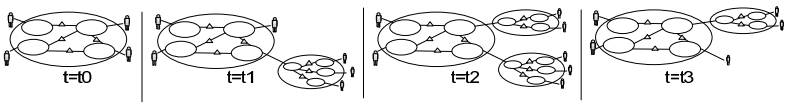


Fig. 7. The snapshots of an organisation hierarchy that scale-out and scale-in

3.4 Capturing Commonalities and Variations

One of the important benefits of supporting scalability via the organisational approach is the ability to capture commonalities and variations of business services and their relationships (Req2). This allows binding services with slightly varying business functionalities and relationships adding some flexibility in service selection. In an organisation hierarchy, the contracts of higher organisations capture the commonalities while the variations are captured in the lower organisations as shown in Fig. 8.

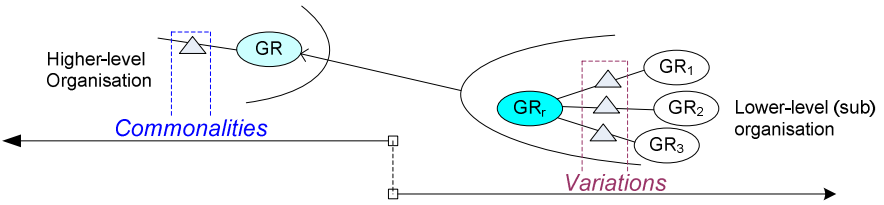


Fig. 8. Capturing commonalities and variations

It should be noted that we project *Interaction Terms* in an automated manner (as RID), yet did not similarly project the *Facts* and *Rules*. The rationale behind this decision is the major differences in corresponding usages. The *Interaction Terms* could be seen as the channels for a message to be passed from one player to another. The identification and propagation of RID to lower-level organisation ensures a smooth end-to-end passage. On the other hand *Facts and Rules* represent the current contract state and how the interactions are evaluated. There is no point of propagating the Facts and Rules of a contract of a higher-level organisation to a contract of the lower-level organisation unless there is a variation of Facts and Rules. An evaluation of one contract along the path of message is sufficient unless there are variations in evaluations. If there are such variations, they should be captured within the contracts of the lower-level organisations of the hierarchy.

To elaborate, consider the snapshot of the organisation hierarchy in Fig. 5. Suppose a message being sent from the currently bound CC service, e.g., *EasyCall*, to one of the repair services. First, the message passes from CC to GR via the contract CC-GR. Then the message is delivered to the sub-organisation which plays the role GR. The router role GR_r routes the message according to defined Routing Strategy, e.g., to GR_2, (played by *BestRepair*) via the contract GR_r-GR_2. Throughout the passage from player *EasyCall* to *BestRepair*, the message is evaluated against two contracts CC-GR and GR_r-GR_2. The facts and rules that are common are captured in the CC-GR (in higher level organisation) whilst the variations applicable only to *BestRepair* are placed in the GR_r-GR_2 (in lower level organisation). For example, the fact *AllowedRepairTypes* is a common fact and rule "*assert the repair request conforms to allowed repair types*" is a common rule, hence placed in CC-GR (Fig. 3). On the other hand, the fact, BonusPayPercentage is a specific fact and the rule "Add a bonus pay amount" is a specific rule, hence placed in GR_r-GR_2 as shown in Fig. 9.

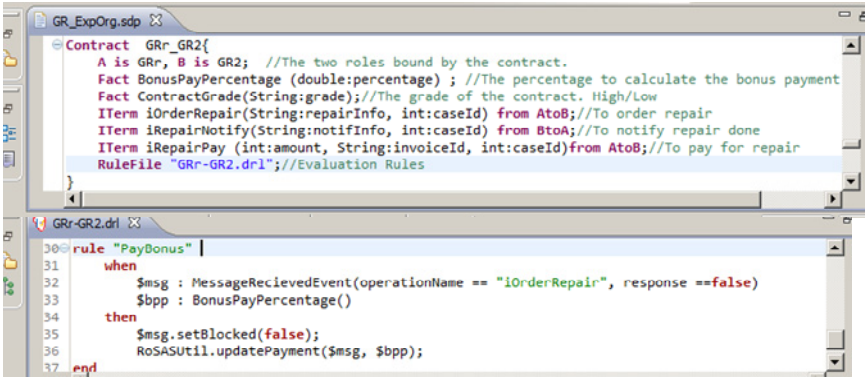


Fig. 9. The contract between GR_r and GR_2

Overall, the organisational approach provides the required modularity to capture commonalities and variations of business service relationships. During runtime contracts of an organisation can be modified to update the relationships.

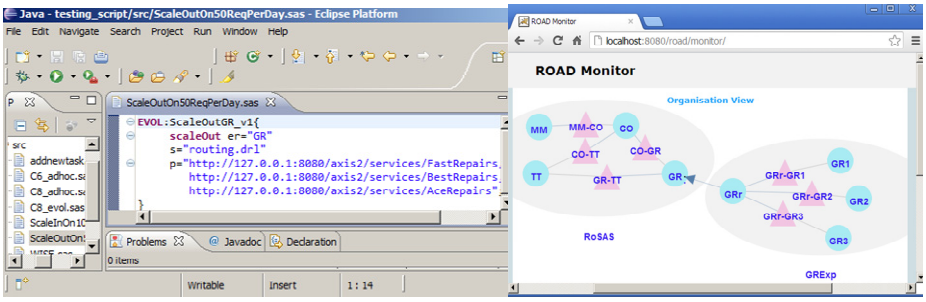
4 Middleware Support

To provide the middleware support for our approach to designing and deploying scalable SaaS applications, we have extended the Role Oriented Adaptive Design (ROAD) framework [12]. ROAD supports the design of adaptable software systems. Its runtime platform (ROAD4WS [13]) extends Axis2 [14], allowing the deployment of adaptive service compositions in a Web service environment. ROAD4WS enables the addition, modification and removal of service composites at runtime. It also provides message mediation and routing capabilities among partner services. Integration with Axis2 allows use of standardised message parsing and delivery protocols, e.g., XML/SOAP and seamless access to other standardised middleware implementations, e.g., WS-Security, WS-Addressing.

The scale-out/in functions have been implemented as high-level operations using the low-level operations of the ROAD framework, e.g., addRole, removeRole, addContract, removeContract [15]. The ROAD framework ensures state consistency in applying these operations e.g., safe completion of transactions [15]. The adaptation scripts containing such operations can be executed immediately or scheduled to be executed upon specific events. For example, the scale-out() operation for GR can be scheduled to be executed upon an event "*more than 50 request per day*".

The contracts are instantiated and maintained as *StatefulKnowledgeSessions* of Drools 5.0 Expert Engine [11]. Such *sessions* can be dynamically inserted with facts (Java objects) and rules (Drools rules) to update the reasoning capabilities.

The interfaces to the roles of the organisations are exposed as WSDL interfaces via Axis2 [14]. These interfaces are automatically created based on the RID (Section 3.3) of a corresponding role [15]. Two types of interfaces generated depending on the direction of Interactions (AtoB or BtoA in Fig. 3). The *Provided Interface* is provided by the SaaS application so that external third party services/clients can send messages. On the other hand the *Required Interface* should be implemented by the third party services so that SaaS application can send messages to them. Tools are provided to model the initial design (Fig. 3), to write the adaptation scripts (Fig. 10-a) as eclipse-based plugins, and to monitor the organisations (Fig. 10-b) through a web interface.



(a) (b)

Fig. 10. Tool support

5 Evaluation

To illustrate the technical feasibility of our approach we have setup a simulation environment based on the motivation example introduced in Section 2.

First we deployed the RoSAS composite and then simulated partner services. We setup the garage (partner) service to have 10s delay to serve requests sequentially (for simplicity only 10s delay is allocated to a repair car). Then we send assistance requests to the RoSAS composite in two different phases, *Low-Frequency* (LF) and *High-Frequency* (HF), where the intervals between two requests in two phases are 15s and 5s respectively. As shown in Fig. 11, the response time kept increasing at the HF phase; (after 20th request) because the rate of requests is higher than the serving capacity of the only available garage service and requests are buffered at the SaaS application.

Then we issue the scale-out command (after the 30th request) to expand the role GR to accommodate two other services (i.e., move the application configuration from Fig. 2 to Fig. 5). Consequently, the response time decreased as now the requests are shared among multiple garage services (here, S = round-robin routing). The decrease is gradual as the requests accumulated in the composite need to be cleared first. The experiment was setup on a closed environment to avoid network delays. We designed the partner services as Web services. The machine had 2.52 GHz Intel Core i-5 CPU with 4 GB RAM. The operating system was 32-bit Windows 7 Home Premium. The servlet container was Apache Tomcat 7.0.8 with Axis2 1.6.2.

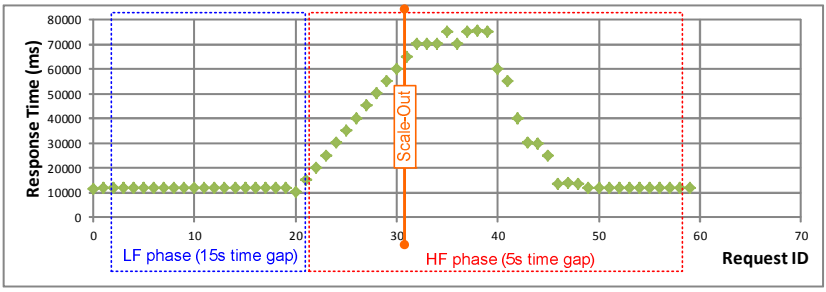


Fig. 11. Evaluation results

We also quantify the average time taken by the middleware to respond to the scale-out and scale-in commands. The Table 1 reports the average time taken to accommodate/remove N number of services. It reveals that even for a large scale-out/in with N=100, it takes approximately 14.3s to complete the scale-out and 0.629s to scale-in. The scale-out is slower compared to scale-in mainly due to rule deployment and instantiation for each new contract. This quantification also ran on the same configuration given earlier. We believe the reported times are reasonable, especially compared with manual reconfiguration, which could have taken much more time to complete.

Table 1. Time to scale-out/in and resume operations

By Number of Roles, N=	1	2	3	4	5	10	100
Average time to Scale-out (ms)	4080	4081	4082	4087	4097	5023	14300
Average time to Scale-in (ms)	21	23	26	29	33	50	629

6 Related Work and Analysis

In this section we compare and analyse our approach against a number of approaches proposed in the past to model SaaS applications in service-oriented environments.

Service Template Markup Language (STML) [16] is a markup language proposed by Zhu et al. to customise and deploy a SaaS application using Model-driven Architecture (MDA). In addition, Sharma et al. [17] too combine the benefits of MDA and SOA to build SaaS applications. While both these approaches provide a technology agnostic methodology to build SaaS applications, there is little attention paid to achieving the scalability and variability requirements of the generated SaaS application instance. For example, there is neither special support for scalability of the generated SaaS service in [16] nor for the transformed PSM in [17]. A new variation requires a re-generation.

Le et al. [6] proposes to model the business objectives and constraints and relate them to the problem of elasticity of business services. While the approach provides a methodology to correlate the non-functional properties to provide elasticity, it does not provide a specific architectural support to scale-out/in the SaaS application.

One of the obvious solutions to SaaS scalability is to use Grid technologies to build SaaS applications because of its ability to provide computing power on demand. For example, GridSaaS [8] is a grid-enabled and SOA-based SaaS application platform that supports the creation of SaaS applications by harnessing existing shared foundational services, e.g., data integration services, authorisation services. While this approach allows sharing of the services, it lacks support for scale-out/in a SaaS application by integrating services with varying capabilities as supported by ROAD4SaaS.

Service Component Architecture (SCA) provides an assembly model to compose heterogeneous applications [18]. In addition, there is an explicit representation of components providing the required abstraction. However, the *component-references* [18] in SCA lack the support to explicitly capture the complex and heterogeneous business service relationships compared to the rich support provided by the *contracts* in our approach.

Another solution is to use an ESB-based (Enterprise Service Bus) approach to model SaaS applications. For example, Cloud Service Bus [19] is an ESB-based approach proposed to integrate different software services into a SaaS platform. While the approach benefits from the inherent advantages of ESB such as dealing with the heterogeneity among services and consumers, again little attention is paid to capturing the commonalities and variations of business services and their relationships.

Hennig et al. [7] propose a scalable service composition approach using the Binary Tree Parallelisation technique. While the approach is capable of harnessing the increased performance of multi-core architecture, the approach does not capture the business relationships among the partner services. Similarly, the proxy based approaches such as TRAP/BPEL [20] helps to scale-out an application instance. However, it does not capture commonalities and variations among business services.

In our previous work we have proposed a multi-tenant architecture to model SaaS applications [21] that allows defining multiple business processes upon a single application instance designed as a business service composition. While that work allows a single application instance being shared among multiple tenants with varying requirements, it lacked the support to scale-out/in the application operations as the demand

fluctuates. In this work we overcome this limitation by adopting and further extending the organisational approach. In comparison to the existing approach, ROAD4SaaS provides a novel design methodology that supports scale-out/in while explicitly capturing the commonalities and variations of partner services and their relationships, which is very important in composing business services to design SaaS applications.

A summary of the comparative analysis of the related works is given in Table 2. Overall, compared to the existing approaches the ROAD4SaaS approach provides a system designer/engineer with the capability to closely capture its heterogeneous business environment in a way that it is possible to scale-out/in the SaaS application that compose business (partner) services. The key characteristics behind this advantage are the extensibility as well as the explicit representation of service relationships supported by the organisational design.

Table 2. A summary of the comparative analysis of the related works

Approach	[16]	[17]	[6]	[8]	[18]	[19]	[7]	[20]	[21]	ROAD4SaaS
Req1	-	-	~	+	+	+	+	+	-	+
Req2	-	-	-	-	-	-	-	-	~	+
Req3	-	-	~	+	+	~	-	+	+	+

+ Supported, - Not Supported, ~ Limited Support

7 Conclusion and Future Work

In this paper we have presented a novel methodology and middleware platform, ROAD4SaaS, to design and deploy scalable SaaS applications that integrate business services. We have analysed the differences in addressing the scalability issue related to composing business services and importance of supporting their commonalities and variations as part of the solution. A service composition is treated as having a hierarchy of organisations that explicitly captures the partner business services and their relationships. The organisation hierarchy can grow/shrink to accommodate more/fewer partner services as the demand for the application changes. It also supports the representation and management of commonalities in business relationships at the higher levels of the hierarchy while allowing variations to be captured at the lower levels. This provides a better modularity as well as a clear separation of concerns in the application design. The middleware and tool support is provided to achieve the scalability in a manual or automated manner.

We are currently developing a graphical programming tool that will allow the developer/organiser to adapt a visual runtime representation of an organisation and organisational hierarchy, rather than using the current script-based approach to dynamically change roles, contracts and bindings.

Acknowledgments. This research was partly supported by the Smart Services Cooperative Research Centre (CRC) through the Australian Government's CRC Programme (Department of Industry, Innovation, Science, Research & Tertiary Education).

References

1. Campbell-Kelly, M.: The rise, fall, and resurrection of software as a service. Communications ACM 52, 28–30 (2009)
2. Laplante, P.A., Jia, Z., Voas, J.: What's in a Name? Distinguishing between SaaS and SOA. IT Professional 10, 46–50 (2008)
3. Suleiman, B., Sakr, S., Jeffery, R., Liu, A.: On understanding the economics and elasticity challenges of deploying business applications on public cloud infrastructure. Journal of Internet Services and Applications 3, 173–193 (2012)
4. Agrawal, D., El Abbadi, A., Das, S., Elmore, A.J.: Database Scalability, Elasticity, and Autonomy in the Cloud. In: Yu, J.X., Kim, M.H., Unland, R. (eds.) DASFAA 2011, Part I. LNCS, vol. 6587, pp. 2–15. Springer, Heidelberg (2011)
5. Amazon Auto Scaling, `http://aws.amazon.com/autoscaling/`
6. Lê, L.S., Truong, H.L., Ghose, A., Dustdar, S.: On Elasticity and Constrainedness of Business Services Provisioning. In: Proceedings of the 2012 IEEE Ninth International Conference on Services Computing, pp. 384–391. IEEE Computer Society (2012)
7. Hennig, P., Balke, W.T.: Highly Scalable Web Service Composition Using Binary Tree-Based Parallelization. In: 2010 IEEE International Conference on Web Services (ICWS), pp. 123–130 (2010)
8. Yong, Z., Shijun, L., Xiangxu, M.: GridSaaS: A Grid-Enabled and SOA-Based SaaS Application Platform. In: IEEE International Conference on Services Computing, SCC 2009., pp. 521–523 (2009)
9. Beugnard, A., Jean-Marc, J., Plouzeau, N., Watkins, D.: Making Components Contract Aware. Computer 32, 38–45 (1999)
10. Kapuruge, M., Han, J., Colman, A.: Representing Service-Relationships as First Class Entities in Service Orchestrations. In: Wang, X.S., Cruz, I., Delis, A., Huang, G. (eds.) WISE 2012. LNCS, vol. 7651, pp. 257–270. Springer, Heidelberg (2012)
11. Amador, L.: Drools Developer's Cookbook. Packt Publishing (2012)
12. Colman, A.: Role-Oriented Adaptive Design. PhD Thesis, Swinburne University of Technology, Melbourne (2007)
13. Kapuruge, M., Colman, A., King, J.: ROAD4WS – Extending Apache Axis2 for Adaptive Service Compositions. In: IEEE International Conference on Enterprise Distributed Object Computing (EDOC), pp. 183–192. IEEE Press (2011)
14. Jayasinghe, D.: Quickstart Apache Axis2. Packt Publishing (2008)
15. Kapuruge, M.: Orchestration as Organisation. PhD Thesis, Swinburne University of Technology, Melbourne (2012), `http://is.gd/z9fgzQ`
16. Xiyong, Z., Shixiong, W.: Software Customization Based on Model-Driven Architecture Over SaaS Platforms. In: International Conference on Management and Service Science, MASS 2009, pp. 1–4 (2009)
17. Sharma, R., Sood, M.: Modeling Cloud Software-As-A-Service: A Perspective. International Journal of Information and Electronics Engineering 2, 238–242 (2010)
18. Chappell, D.: Introducing SCA (2007), `http://is.gd/Cj3Mab`
19. Aobing, S., Jialin, Z., Tongkai, J., Qiang, Y.: CSB: Cloud service bus based public SaaS platform for small and median enterprises. In: 2011 International Conference on Cloud and Service Computing (CSC), pp. 309–314 (2011)
20. Ezenwoye, O., Sadjadi, S.M.: TRAP/BPEL: A Framework for Dynamic Adaptation of Composite Services. In: WEBIST 2007, Barcelona, Spain (2007)
21. Kapuruge, M., Colman, A., Han, J.: Achieving Multi-tenanted Business Processes in SaaS Applications. In: Bouguettaya, A., Hauswirth, M., Liu, L. (eds.) WISE 2011. LNCS, vol. 6997, pp. 143–157. Springer, Heidelberg (2011)