Achieving Multi-tenanted Business Processes in SaaS Applications

Malinda Kapuruge, Alan Colman, and Jun Han

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

Abstract. With the emergence of Cloud Computing and maturity of Service Oriented Architecture (SOA), the Software-as-a-Service (SaaS) delivery model has gained popularity, due to advantages such as lower startup cost and reduced time to market. A SaaS vendor owns and takes the responsibility of maintaining a single application for multiple clients/tenants who may have similar but also varying requirements. Business process modeling (BPM) approaches can be used to package service offerings to meet these varying requirements on a shared basis. However the customizations in those business processes can be challenging. In this paper we discuss the challenges arising from single-instance multi-tenancy, and present our approach to defining business processes in SaaS applications to address those challenges.

Keywords: SaaS, Cloud, SOA, BPM, Multi-tenancy.

1 Introduction

Software-as-a-Service (SaaS) is a software delivery model that lets a customer (tenant) remotely utilize hosted software over the internet and pay according to a certain subscription package [1, 2]. SaaS is being increasingly adopted due to its lower startup costs and higher reliability compared to the *on-premise* software development and delivery model. In contrast to the *on-premise* model, a SaaS vendor [3] owns, hosts and maintains the software application and the underlying infrastructure in the hosting environment, freeing the tenants from those burdens [4].

It has been observed that SOA and SaaS are very closely related architectural models [3]. Service Oriented Architecture (SOA) plays an important role in realizing the SaaS concepts in the enterprise architecture[5]. In a SaaS delivery model the SOA principles can be used to easily integrate existing available services to broaden business offerings, especially when the SaaS vendor alone cannot provide all the required functionalities. Also, in order to specify the order in which the services are delivered and how the dependent services should be invoked, a Business Process Modeling (BPM) mechanism is typically employed. The advantages of BPM such as automated enactment, re-design and verification can be leveraged in delivering Software-as-a-Service simultaneously to multiple tenants in native multi-tenant environments[6].

Consequently, there is an interest in using BPM languages to orchestrate service delivery in SaaS applications. However, *single instance multi-tenant*(SIMT)

A. Bouguettaya, M. Hauswirth, and L. Liu (Eds.): WISE 2011, LNCS 6997, pp. 143–157, 2011.

applications[7] brings additional challenges to BPM. A SaaS vendor has to capture the commonalities in process definitions, maintain different variations as required by tenants, and achieve effective isolation in modifications to protect the integrity of tenants' requirements and the application as a whole. Most work on meeting similar challenges has occurred in providing virtual individualized data service to tenants. However defining workflows and processes is also a key aspect of SOA, and one that needs addressing if SOA systems are to be deployed as *true* multi-tenanted SaaS applications. In this paper we discuss the relevant challenges in detail and introduce our approach, *Serendip4SaaS* to defining business processes in SaaS applications to address these challenges.

In order to analyze the problem, a motivational scenario is given in Section 2. We will also highlight the challenges and requirements for BPM in SIMT applications. The section 3 gives an overview of our approach. In Section 4, we discuss how our approach is capable of meeting those challenges. Section 5 introduces our prototype implementation. Related works are analyzed in Section 6. Finally, we conclude the paper in Section 7.

2 Problem Analysis

In order to understand the problem and motivate the discussion, we first present an expository scenario. We will then discuss the challenges of BPM in multi-tenant SaaS applications based on the presented scenario.

2.1 Motivation

A Road Side Assistance service benefits motorists by providing emergency assistance when there is a car breakdown. Software systems are being used to coordinate the activities such as towing, repairing etc. Possible businesses that need such a service may include insurance companies, car sellers, and travel agents, providing road side assistance as a value added service to attract clients. For businesses whose core-business is not Road Side Assistance, running and maintaining such a software system and the infrastructure is an onerous task. In such situations it is cost effective and efficient to use an external service and the underlying infrastructure of such kind on a subscription basis [8].

In order to meet this market need, *RoSaaS.com* provides road side assistance as a service. *RoSaaS* (Road Side Assistance as a Service) provides different service packages for its customers depending on their requirements. RoSaaS as the SaaS vendor contracts various third party service providers, including garages (GR), tow car (TC) services, taxis (TX) and paramedics (PM). Other service providers such as case handling officers (CO) could be either handled internally by RoSaaS or outsourced as appropriate. The RoSaaS software handles the complexity of the underlying operations such as requesting, meditating and monitoring third-party and internal services and binding/unbinding service endpoints etc. However, tenants (e.g. car-sellers, travel agents) can request customizations due to changing business goals.

An overall view of the RoSaaS business model is given in Fig. 1(a). The lowest layer represents the internal and external (third party) service providers/collaborators. Then *RoSaaS.com* needs to integrate these third party services according to a well-defined choreography. Such choreography defines an *acceptable* ordering and

scheduling of activities in the composition. Based on the offerings of the *RoSaaS* platform different subscriptions/customizations can be delivered to its subscribing customers. The top layer represents the end users, i.e. motorists (MM) that ultimately use the road side assistance, e.g. a traveller or an insurance policy holder.

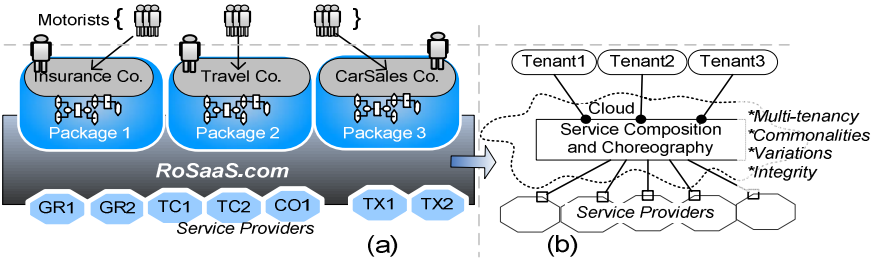

Fig. 1. RoSaaS: (a) Business model, (b) SaaS realization (application architecture)

2.2 Challenges and BPM Requirements

SaaS vendors as business entities have to depend on third party service providers to provide certain functionalities (e.g. towing and repairing) as they alone might not be able to meet all customer requirements. Such dependencies are becoming common as more and more enterprise applications are now exposed and delivered as services [6, 9]. The SaaS realization of the RoSaaS business model is given in Fig. 1(b). The service composition is required to combine the various third-party lower level services, and deliver application level services in a multi-tenant environment, on demand.

In ***multi-tenancy*** one application instance is utilized by all its tenants [7]. Each tenant interacts with the system as if they are the sole user [6]. A tenant can request modifications to its *package* to suit their changed business objectives. However, these modifications are applied on a single shared application instance. Subsequently modifications could be available to other tenants who use the same application instance. In some cases this might be a necessity, e.g. applying a patch/upgrade. However, in other cases, modifying a common application instance can challenge the integrity of the application instance, compromising the objectives of RoSaaS and other tenants. For example, a request by the tenant, *CarSellerCo* to add additional pre-condition for towing, might delay the execution and thereby hinder the objectives of another tenant such as *InsuranceCo* who utilize the same application instance. Therefore achieving effective isolation in process modifications is a must.

One naïve solution to achieve isolation of process modifications is to allocate dedicated process definitions for each and every tenant, i.e. similar to lower level of the *maturity model* [10]. However, it should be noted that tenants of a SaaS application have common business interests. Hence, there can be *significant overlapping* in those business processes. Having separate process definitions, leads to code duplication. Such duplication deprives the SaaS vendor from exploiting the benefits of SIMT. The SaaS vendor has to apply modifications repeatedly to these process definitions, which is not efficient and could be error prone. Therefore the BPM approach and the language utilized in RoSaaS should be able to ***capture commonalities*** in behavior. These common *behaviors* or *process segments* can then be reused to define complete *road side assistance business process definitions* as part of the software packages for

individual tenants. As an example, the *towing behavior* could be common to all customer processes and consequently the code/script can be shared among different process definitions.

Although *capturing commonalities* is essential and beneficial to any SaaS vendor, it is allied with two accompanying challenges. Firstly, while the tenants have common business requirements, these requirements may slightly vary in practice. As such, RoSaaS cannot assume a common code/script can continue to serve all the tenants in the same manner. As an example, even though the abovementioned *Towing Behavior* is common to all customers, during design time or runtime a tenant might request a change in pre-conditions to start towing. Therefore the BPM approach should be able to ***allow variations***, while capturing the commonalities.

Secondly, capturing commonalities might lead to ***invalid boundary crossings***. To elaborate, suppose that tenant *CarSellerCo* requests a modification to the *Towing Behavior* of a road side assistance process. However, this modified business process might violate a business requirement of another tenant such as *InsuranceCo*, because the *InsuranceCo*'s package shares the same behavior. Moreover, as the RoSaaS is a service composition, such changes might lead to violations to the business models of some of its collaborating third party service providers such as *Tow cars* and *Garages*.

To support business processes in single-instance multi-tenant and service oriented applications, in general, it is essential that the following requirements are fulfilled.

Req 1. *Commonalities* in behavior of the SaaS application and its service delivery should be captured to reduce redundancy.

Req 2. *Variations* in behavior should be allowed as tenant requirements are similar but not the same.

Req 3. *Invalid boundary crossings* should be prevented. In this sense, the BPM approach should be capable of identifying the impact of a change from one tenant on both other tenants and the underlying collaborator services.

3 The Approach

In this section we will first give an overview of our process modeling approach. Then we will discuss how the above objectives are achieved in the next section by providing more details.

In order to address the above mentioned requirements, we propose an *organizational* approach to modeling business processes in SaaS applications. We envision a SaaS application instance as an **organization** that provides an abstraction of the underlying collaborating services. Then we define acceptable *behaviors* (e.g. Towing, Repairing) of the organization in a *declarative* manner on top of this organizational structure. These declarative behaviors are then used to construct a business process (view) in the package for each subscribing tenant in the form of an Event-driven Process Chain (EPC) [11].

As shown in Fig. 2, we identify three layers in the SaaS application design. The lowest layer defines the service-service interactions in the application system. We use the ROAD [12, 13] design concepts such as contracts and roles to modularize these interactions in a declarative manner. ROAD allows the definition of **roles** and the **contracts** between roles in a composition/system. A role is a position description and should be *played* by a business service like *web services* hosted in the Garages, Tow

cars, Taxis or even a client application in Case Officers' mobiles. The contracts among roles capture the mutual obligations and responsibilities among these roles in terms of allowed **interactions**. A Role player performs **tasks** by interacting with other role players[14]. As an example, a task in the road side assistance process is *Tow()*, which is an execution step carried out by a bound *towing service*. The towing service needs to interact with other role players, e.g. case officer, garage services, who perform tasks too. However the execution of these tasks needs to be ordered.

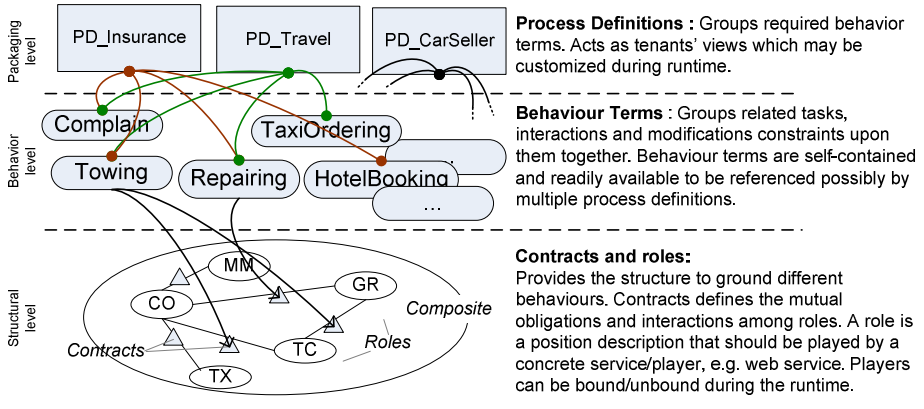

Fig. 2. Different levels of the RoSaaS organization

To achieve that, as shown in Fig. 2, we define different behaviors which we call **behavior terms** on top of structured interactions in the lower organizational layer. Each behavior term specifies an acceptable behavior (ordering of tasks) of the organization. For example, *when towing is required, the case officer (CO) should notify the tow car (TC); upon that notification, the TC should perform towing*. These behavior terms are self-contained and group related tasks that need to be performed by services playing roles in the organization. They also capture any constraints (**behavior constraints**) that govern the order of execution of tasks. This allows ensuring that specific orderings of tasks are not violated during runtime customizations.

At the top layer, **process definitions** make use of such predefined behavior terms. A process definition (a tenant's view of the application or organization behavior) is allocated to each tenant's package. Multiple process definitions can share the same behavior term. These views can be customized at runtime to suit tenants' changed requirements. However such changes are applied in the same organizational structure (single application instance) and should respect the constraints defined in related behavior terms. A business process of a particular tenant can also add additional constraints (**process constraints**) to ensure the goals of the tenant are not violated upon runtime customizations.

Fig. 3 presents a meta-model summarizing the concepts we discussed above.

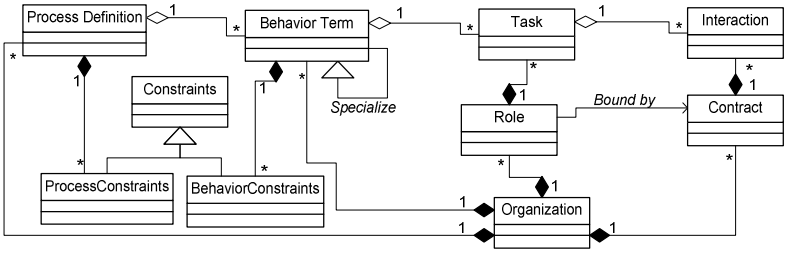

Fig. 3. The Serendip4SaaS meta-model

4 Addressing Requirements

In this section we demonstrate how our approach fulfills the three requirements, defined in section 2.2, of BPM in multi-tenant environments, namely, capturing commonalties, allowing variations and preventing invalid boundary crossings.

4.1 Capturing Commonalities (*Req 1*) via Behavior Modeling

Tenants of a SaaS application will share some degree of common interest. Therefore the business processes defined to serve the tenants naturally show a *significant overlapping* in the required tasks. As in our scenario, tasks and their ordering associated with car *towing* and *repairing* can be common to many process definitions. As mentioned before, defining dedicated multiple business processes for each tenant can lead to redundancy and need to be avoided. In our approach, behavior terms provide the basis for *modularity* and *re-use* and thereby capture the commonalities across tenants' business processes.

```
BehaviorTerm Towing{
    Task  SendTowReq{
          EPpre TowReqd;
          EPpost TowReqSent & PickupLocKnown;
          PerfProp 2h;
          Roblig CO;
    };
    Task  Tow{
          EPpre PickupLocKnown &  DestinationKnown;
          EPpost CarTowed ;
          PerfProp 24h;
          Roblig TC;
     };
    Task  PayTow{
          EPpre CarTowed & TowAcked;
          EPpost TCPaid;
          PerfProp 2h;
          Roblig CO;
     };
     //More tasks ...

    Constraint c1: (CarTowed>0)->(TCPaid>0);
    Constraint c2: (TowReqSent>0)->[4h,24h](CarTowed>0);
    //More constraints...
};
```

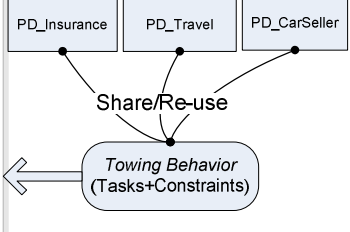

Fig. 4. A sample behavior term

The sample shown in Fig. 4 presents three tasks (*SendTowReq, Tow, PayTow*) associated with the *Towing* behavior, possibly shared by multiple process definitions.

Each task specifies certain attributes, which includes pre-conditions (EPpre), post-conditions (EPpost), performance properties (PerfProp) and Obligated role (Roblig).

For example, the above *Tow* **task** specifies "*when the pickup location is known and the destination is known (=EPpre), the TC role is obliged to perform task Tow within 24 hours. Once the towing is complete, it will trigger events CarTowed and TowAcked(=EPpost)*". Note that the task dependencies are represented via events. For example, task *SendTowReq triggers* event *PickupLocationKnown*, which is a pre-condition of task *Tow*. This means task *Tow* should happen after task *SendTowReq*. Similarly, task *PayTow* happens after task *Tow*.

A behavior term also specifies *modification constraints* on how these tasks should be carried out. We use TCTL[15] to specify such constraints. As an example the first constraint specifies that *"every CarTowed event should eventually be followed by TCPaid event"*. We will discuss the use of constraints in section 4.3 in detail.

Two sample process definitions *PD_Insurance* and *PD_Travel* are shown in Fig. 5. A process definition can refer to a behavior term using the attribute `BehaviorTermRef`. As shown, both the *PD_Insurance* and *PD_Travel* re-use the behavior terms *Towing* and *Repairing*, because both tenants *InsuranceCo* and *TravelCo* require to provide towing and repairing for their customers i.e. policy holders and travellers. Specifying such re-usable behavior terms allows capturing the commonalities in different processes.

Also each process definition may specify its own process level constraints using attribute `Contraint`. As an example the *PD_Insurance* specify specific constraint that *an end user request through InsuranceCo should be fulfilled within 5 days*. Such process level constraints protect tenants' goals (*See* section 4.3).

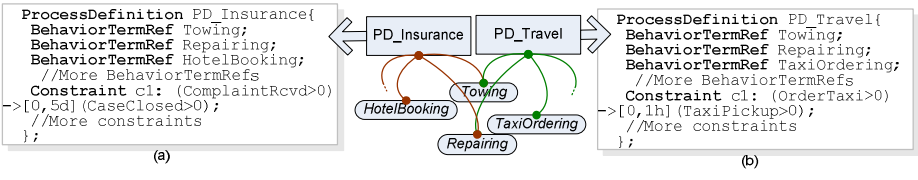

Fig. 5. Sample process definitions

When a process definition group multiple behavior terms, the framework dynamically constructs an EPC graph[11] to present the view of the process definition for the corresponding tenant. The EPC graph is constructed by merging behavior terms using the event dependencies. Fig. 6 shows a section of a complete EPC graph. As shown, the common events (e.g. *CarTowed*) of two tasks are mapped together to construct the process view.

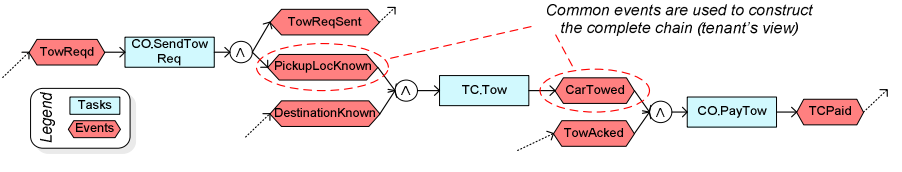

Fig. 6. Process view (EPC) construction from declarative task definitions

In summary, the key characteristic to fulfill *Req1* is the *re-usability* and *modularity* provided by the behavior terms. Furthermore, the *use of events* to define task dependencies helps to achieve loose-coupling among these modularized behavior terms.

4.2 Allowing Variations (*Req 2*) via Behavior Specialization

In SaaS applications it is essential to address the *variations in behavior* as pointed out in section 2.2. As mentioned, process definitions can have specific behavior terms to allow variations, while sharing some common behavior terms. However, it is likely that during runtime these common behavior terms might fail to facilitate the unforeseen variations. As an example, after some time, a tenant (*CarSellerCo*) might require additional conditions to start *tow*; another tenant (*TravelCo*) might require a new *notification* to be sent to the traveller/motorist, once the *tow* is complete.

To support such variations we use *behavior specialization*. In this sense, a behavior term can *specialize* another already defined behavior term to create a new one by specifying the additional properties or overriding existing properties. We call the new behavior a **child** of the already defined **parent**. The parent behavior can either be a *concrete* or an *abstract* (i.e. cannot be packaged/instantiated) behavior term.

Shown in Fig. 7 is a new behavior called *Towing2*, which *specializes* the already defined *Towing* behavior, previously shown in Fig. 4. The child *Towing2* term specifies only those tasks and attributes that would override those of the parent *Towing* term. As shown, the attribute **EPpre** has been changed to delay the *tow* task until the taxi picks up the motorist as required by the tenant *CarSellerCo*.

By specializing (extending) *Towing*, the new *Towing2* will inherit,

1. All the other attributes specified in task *Tow*. E.g., EPpost, PerfProp, Roblig.
2. All the other tasks specified in *Towing,* e.g., SendTowReq, PayTow.
3. All the constraints specified in *Towing,* e.g., c1, c2.(See Fig. 4).

Now the process definition *PD_CarSeller* can refer to the new behavior term *Towing2* instead of *Towing*. The framework will identify these specializations and recognize the inheritance hierarchy to complete the partially defined tasks and attributes of child behavior term. Then the framework will dynamically build the process view as mentioned in the previous subsection. The variation on the constructed EPC graph due to switch to *Towing2* is shown in Fig. 8.

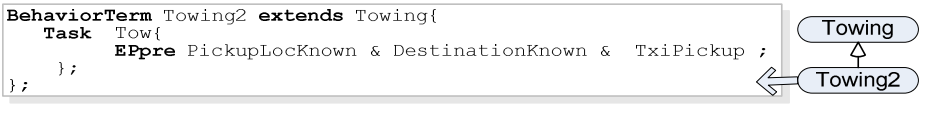

Fig. 7. Specializing behavior (property change of a task)

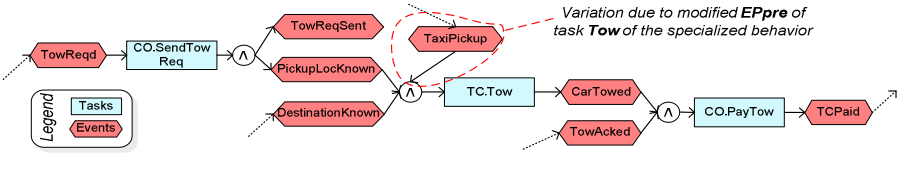

Fig. 8. Variation in *CarSellerCo's* process due to switch to *Towing2*

Apart from specializing attributes of a Task, a behavior term can add additional task(s) too. For example, if a *TravelCo* service requires an alert to be sent to Motorist upon towing, apart from other usual interactions, a new behavior term *Towing3*, may specify an additional Task *AlertMM* as shown in Fig. 9. Now instead of using *Towing,* the *PD_Travel* may refer to *Towing3*. The variation is shown in Fig. 10.

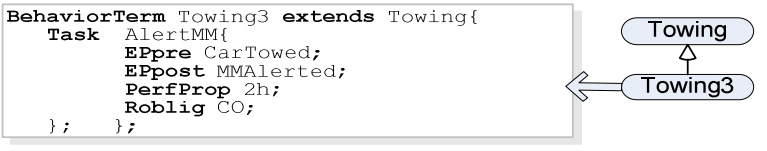

Fig. 9. Specializing behavior (additional task)

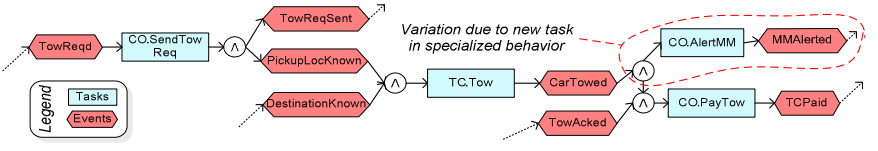

Fig. 10. Variation in *TravelCo's* process due to switch to *Towing3*

The advantage of such specialization is that a parent behavior can capture the *commonalities* of behavior whilst the children specialize them to capture the *variations*. This mechanism keeps the redundancy in the script to a minimum (e.g., behavior terms *Towing2* and *Towing3* only specified additional properties). Also, since the child inherits all the properties from the parent, the modification constraints defined in the parent can act as a guard to ensure the variation does not lead to any violations (see next section). Furthermore, such specializations can be used in a similar fashion to keep multiple variations consistent over variations in underlying collaborating services.

In summary, the key to achieve specialization is the declarative nature of the behavior terms. Rather than using imperative workflow modeling languages such as EPC [11], we use a declarative language, which made it possible to extend/override the declaratively specified properties. Nonetheless, for visualization purposes we dynamically construct the workflow in the form of an EPC graph by merging the referenced behavior terms together, specialized or otherwise.

4.3 Preventing Invalid Boundary Crossing (*Req 3*) via Two-Level Constraints

Tenants might demand modifications to their process views as shown earlier. However, these modifications are applied in a single application instance. Such a modification might potentially violate a goal of another tenant or even an underlying collaborator. We call them *invalid boundary crossings*. To prevent such *invalid boundary crossings*, our language provides two levels of constraint specifications.

Behavior level constraints are defined in behavior terms, e.g., in *Towing*, irrespective of the enclosing business processes. Once a process definition refers to the behavior term as shown in Fig. 5, the integral modification behavior constraints (both specified and inherited) are applicable. The goal is to ensure that the underlying service-service collaborations are not affected by the customizations to the processes.

Process level constraints are defined in business processes, e.g., in *PD_Insurance*. The goal is to ensure that the modifications/patches to underlying service-service collaborations do not affect the goal of the tenant, e.g., *InsuranceCo*.

For example, in Fig. 4, the behavior term *Towing* has a constraint: "*every CarTowed event should eventually be followed by TCPaid event*". But in some situations additional constraints need to be defined in a much broader *Process* level, across multiple aggregated behavior terms. For example in Fig. 5 the process definition *PD_Insurance* defines a constraint: *"the total time from car-is-towed to car-is-repaired needs to be within 5 days"*. Here the towing activities and events are defined in a behavior term *Towing* while car repair activities and events are defined in another behavior term *Repairing*.

Suppose the tenant *CarSellerCo* wants to add an additional event to the pre-condition of *tow* task. However this modification results in an alteration in *Towing* behavior shared by another tenant, e.g., *InsuranceCo*. But new pre-condition might delay the start of *tow* task and thereby increase the estimated time to complete a case, possibly violating *InsuranceCo's* goals. Detecting such invalid boundary crossings without an automated validation can be a time consuming and a tedious task.

Therefore, as shown in Fig. 11, prior to applying a modification requested by a tenant to a behavior term β, two different types of validations are carried out by the framework to analyze the impact of the modification.

Validation 1. All the constraints defined in the behavior β are not violated.

Validation 2. All the constraints defined in process definitions (*sharing process definitions*) that share behavior term β are not violated.

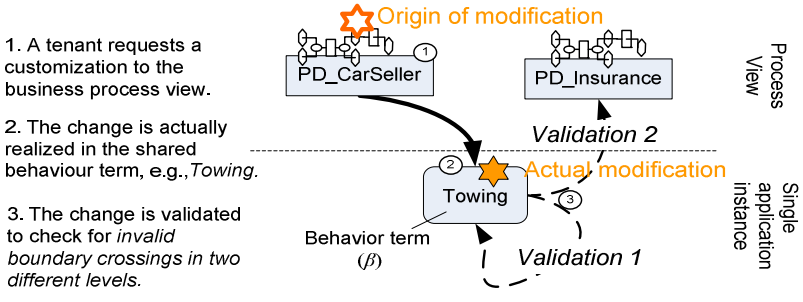

Fig. 11. Two level constraint validation

To formally validate the integrity, we convert the constructed EPC into a Time-Petrinet (TPN) [16] according to the translation rules introduced by van der Aalst et al.[17]. A TPN is a directed bipartite graph that can specify firing intervals of transitions. Places, transitions, firing intervals in a translated TPN are analogous to the events, tasks and estimated time for completing tasks. We use TCTL (CTL with time properties)[15] to specify the constraints. TCTL properties has been used to validate TPNs previously [16]. After mapping the elements (i.e. events, time units) defined in the constraints into the *places* and *firing intervals* of generated TPN, we validate whether the generated Petri-net conforms to the defined set of constraints. To achieve this, we have implemented a wrapper module for *Romeo on-the-fly model checker* [18] which performs the transformation and validation on the fly. If the validation has a

negative outcome, the change is rejected and the violated constraints are shown. Such a validation will occur every time a change is made to the behavior terms to ensure the integrity of the SaaS application instance is not compromised. Upon such modification failures, RoSaaS designer might take further actions, such as looking for a possibility of relaxing a constraint or specializing the behavior term to the client so that other tenants are not affected.

In summary, the way we incorporate temporal constraints into *behavior terms* and *process definitions* means *only the relevant set* of constraints will be considered[14]. Such well-scoped validation reduces the number of constraints that need to be validated without unnecessarily restricting the possible modifications to behaviors. This is an improvement compared to setting up a global set of constraints (applicable for all the processes and collaboration behaviors) to protect the integrity of an application.

5 Implementation

Fig. 12 presents an overview of the implementation framework for our approach to realizing multi-tenanted business processes. Initially the SaaS designer/vendor defines the allowed behaviors using a set of behavior terms at design time. During the runtime, tenants can request the customizations using the *Process Customization and Visualization Tools*. However, the actual customization is realized in the *Model Provider Factory (MPF)*, which maintains all behavior terms including their specializations in a single runtime. The *SaaS designer* too can apply the patches and upgrades to the core behaviors apart from constructing and modifying the views during the runtime. A screenshot of the GUI of *Designer tool* is given in Fig. 13.

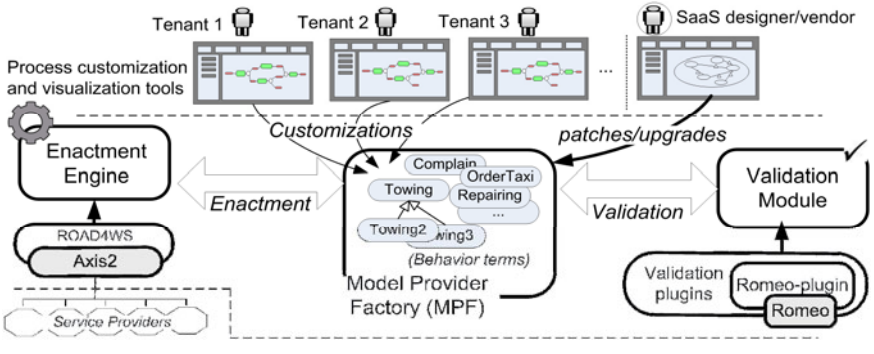

Fig. 12. Implementation Architecture

The *MPF* uses the *Validation Module* to validate the correctness of the defined behaviors and process definitions upon modifications, i.e. both vendor's patches and tenants' customizations. As mentioned in section 4.3, we use the *Romeo on-the-fly model checker*[18] to validate the TCTL[15] constraints against the generated TPN[16]. If a modification violates some constraints, i.e., goals of other tenants or underlying collaborators, it will be rejected. Then the issue will be escalated to the RoSaaS designer pinpointing the affected behavior terms and violated constraints.

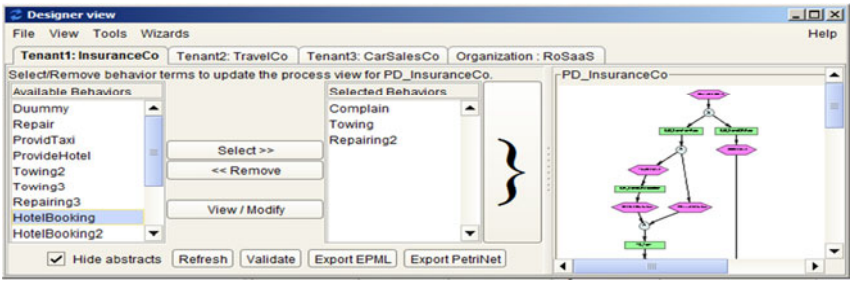

Fig. 13. A screen shot of the Designer Tool of the prototype

The *Enactment Engine* uses the *behavior terms* as grouped by *process definitions* to enact and maintain process instances (as parts of the single SaaS application). For example, the engine will enact a process instance of the definition *PD_Insurance* (Fig. 5) to handle a case of a *Motorist* who is a policy-holder of *InsuranceCo*. We use *ROAD4WS* [13], which is the web service realization of ROAD[12] and an extension to Apache Axis2, for binding and interacting with third party service providers such as web services hosted in garages, tow-car companies etc.

6 Related Work

WS-BPEL [19, 20], which is considered the de-facto standard for web service integration, is an obvious candidate for process centric SaaS applications. However, the imperative nature of WS-BPEL limits its adaptability and maintainability, which are prerequisite for SaaS integration/applications. For example, in a multi-tenant environment, supporting each tenant with a dedicated BPEL process can lead to redundancy and potential inconsistency across the processes, compromising the adaptability and maintainability of the SaaS application. One solution for this issue is to use explicit opaque tokens in abstract BPEL[20]. However these abstract descriptors need to be converted to executable BPEL processes prior to enactment. Once enacted, the concrete process cannot be changed dynamically. Therefore the solution does not provide the required agility of adaptation as required by a SaaS application.

More recently a few approaches have been suggested to overcome such issues. Mietzner et al. proposes a customization process for service-oriented SaaS applications via variability descriptors [21]. In this sense, a provided template is further customized based on tenant requirements. Later these templates can be transformed into BPEL processes models. Similarly the VxBPEL language extension for BPEL attempts to allow a process to be customized based on variability points [22]. However, the variability points are fixed via the appropriate parameters during runtime. Furthermore, there are also many aspect oriented approaches[23, 24] that have taken similar paths to address the issue.

These *template-based* or *aspect-oriented* approaches help to identifying the commonalities and variations to a certain degree. For example, the aspects/rules viewed via point-cuts in the AO4BPEL [23] represents the volatile part, while the abstract process which defines the point-cuts represents the fixed part. Similarly in [21], the variability points represents the volatile part while the provided template

represents the fixed part. However, these approaches suffer from a ***common weakness***, i.e., they assume that the fixed part and the volatile part can be well-identified at the design time. In contrast, our approach does not rely on such an assumption. Rather than differentiating the volatile and fixed at the design time, we use concepts such as *behavior specialization* to further customize the business processes. As such, the customization on definitions is not limited the design time and can be carried out at runtime. This is beneficial for SaaS vendors and tenants, who might not be able to foresee all the variations upfront.

Inheritance has been used earlier in defining business processes[25]. However, our use of inheritance is for specializing self-contained behaviors defined within an organization, providing reusability and modularity at a finer-grained level. This is in clear contrast to using inheritance for a complete workflow. A SaaS vendor can serve the tenants with different variations of behavior by selecting the specialized behavior terms to form processes suitable for specific tenants on-demand.

Note that the use of temporal constraints is not new[16]. Nonetheless the way we modularize such temporal constraints using the modularization and controlled adaptation provided by the organizational structure in order to define and enact business processes for multiple tenants is a key difference between our approach and the rest. This allows customizations to business processes on demand without compromising the integrity or maintainability of the SaaS application instance. The vendor can utilize the existing already defined behaviors to create customizations by changing or *specializing* them to adjust to latest business and operating conditions but with the necessary checking required.

A summary of comparison is given in Table 1.

Table 1. A summary of comparison

Approach / *Feature*	**Kuo [26]**	**Charfi [23]**	**Graml [24]**	**Michiel [22]**	**Grivas [2]**	**Mietzner[21]**	**Serendip 4SaaS**
Customizability	+	+	+	+	+	+	+
Process support	-	+	+	+	-	-	+
Support for single-instance multi-tenancy	-	-	-	-	-	+	+
Capturing commonalities in behavior	-	~	~	~	~	~	+
Facilitating variability in behavior	-	~	~	~	~	~	+
Prevent invalid boundary crossings	-	-	-	-	-	-	+

+ exist, ~ exist with limitations, - does not exist/not applicable

7 Conclusion and Future Work

In this paper we have presented an approach that defines a SaaS application as an organization-based service composition and delivers its service offerings with variations to multiple tenants, via Business Process Modeling (BPM). Such a SaaS application or composition (instance) utilizes third party collaborator services and offers the tenants similar but varying functionalities, while respecting the business rules of all the stakeholders. In particular, we have highlighted how our BPM approach achieves *single-instance multi-tenancy* and meet the requirements of supporting

commonalities and *variations* while preventing *invalid boundary-crossings* between tenant processes when changes are realized. Our approach includes a process modeling language and an implementation framework that helps SaaS vendors and tenants in modeling and managing their applications. The work left for future includes, achieving similar requirements in the data-flow aspects of process modeling and improving the tool support for a better user experience.

Acknowledgments. This work is partly supported by Smart Services CRC, Australia.

References

1. Liang-Jie, Z., Qun, Z.: CCOA: Cloud Computing Open Architecture. In: IEEE International Conference on Web Services (ICWS), pp. 607–616 (2009)
2. Grivas, S.G., Uttam Kumar, T., Wache, H.: Cloud Broker: Bringing Intelligence into the Cloud. In: IEEE 3rd International Conference on Cloud Computing (CLOUD), pp. 544–545 (2010)
3. Laplante, P.A., Jia, Z., Voas, J.: What's in a Name? Distinguishing between SaaS and SOA. IT Professional 10, 46–50 (2008)
4. Waters, B.: Software as a service: A look at the customer benefits. Digital Asset Management 1, 32–39 (2005)
5. Sathyan, J., Shenoy, K.: Realizing unified service experience with SaaS on SOA. In: Communication Systems Software and Middleware and Workshops, COMSWARE 2008, pp. 327–332 (2008)
6. Mietzner, R., Leymann, F., Papazoglou, M.P.: Defining Composite Configurable SaaS Application Packages Using SCA, Variability Descriptors and Multi-tenancy Patterns. In: Internet and Web Applications and Services (ICIW), pp. 156–161 (2008)
7. Chang Jie, G., Wei, S., Ying, H., Zhi Hu, W., Bo, G.: A Framework for Native Multi-Tenancy Application Development and Management. In: Enterprise Computing, CEC/EEE, pp. 551–558 (2007)
8. Campbell-Kelly, M.: Historical reflections. The rise, fall, and resurrection of software as a service. Communications ACM 52, 28–30 (2009)
9. Barros, A., Dumas, M.: The Rise of Web Service Ecosystems, vol. 8, pp. 31–37. IEEE Computer Society, Los Alamitos (2006)
10. Chong, F., Carraro, G.: Architecture Strategies for Catching the Long Tail, MSDN Library. Microsoft Corporation (2006)
11. Scheer, A.-W.: Business Process Engineering: Reference Models for Industrial Enterprises. Springer-Verlag New York, Inc., Secaucus (1994)
12. Colman, A., Han, J.: Using role-based coordination to achieve software adaptability. Science of Computer Programming 64, 223–245 (2007)
13. Kapuruge, M., Colman, A., King, J.: ROAD4WS – Extending Apache Axis2 for Adaptive Service Compositions. In: Enterprise Computing Conference (EDOC). IEEE Press, Los Alamitos (2011)
14. Kapuruge, M., Colman, A., Han, J.: Controlled flexibility in business processes defined for service compositions. In: Services Computing (SCC), pp. 346–353. IEEE Press, Los Alamitos (2011)
15. Baier, C., Katoen, J.-P.: Principles of Model Checking. The MIT Press, Cambridge (2008)
16. Boucheneb, H., Hadjidj, R.: CTL* model checking for time Petri nets. Theoretical Computer Science 353, 208–227 (2006)

17. van der Aalst, W.M.P.: Formalization and Verification of Event-driven Process Chains. Department of Mathematics and Computing Science. Eindhoven University of Technology (1999)
18. Gardey, G., Lime, D., Magnin, M., Roux, O.: Romeo: A tool for analyzing time petri nets. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 418–423. Springer, Heidelberg (2005)
19. Weerawarana, S., Curbera, F., Leymann, F., Storey, T., Ferguson, D.F.: Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More. Prentice Hall PTR, Englewood Cliffs (2005)
20. OASIS: Web Services Business Process Execution Language Version 2.0. (2006), http://docs.oasis-open.org/wsbpel/v2.0/
21. Mietzner, R., Leymann, F.: Generation of BPEL Customization Processes for SaaS Applications from Variability Descriptors. In: Services Computing (SCC), pp. 359–366 (2008)
22. Michiel, K., Chang-ai, S., Marco, S., Paris, A.: VxBPEL: Supporting variability for Web services in BPEL. Information and Software Technology 51, 258–269 (2009)
23. Charfi, A., Mezini, M.: Hybrid web service composition: business processes meet business rules. In: International Conference on Service Oriented Computing, pp. 30–38. ACM, New York (2004)
24. Graml, T., Bracht, R., Spies, M.: Patterns of business rules to enable agile business processes. In: Enterprise Distributed Object Computing Conference, vol. 2, pp. 385–402 (2008)
25. van der Aalst, W.M.P., Basten, T.: Inheritance of workflows: an approach to tackling problems related to change. Theory of Comp. Sci. 270, 125–203 (2002)
26. Kuo, Z., Xin, Z., Wei, S., Haiqi, L., Ying, H., Liangzhao, Z., Xuanzhe, L.: A Policy-Driven Approach for Software-as-Services Customization. In: 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services, pp. 123–130 (2007)