Runtime Evolution of Service-Based Multi-tenant SaaS Applications

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

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

Abstract. The Single-Instance Multi-Tenancy (SIMT) model for service delivery enables a SaaS provider to achieve economies of scale via the reuse and runtime sharing of software assets between tenants. However, evolving such an application at runtime to cope with the changing requirements from its different stakeholders is challenging. In this paper, we propose an approach to evolving service-based SIMT SaaS applications that are developed based on Dynamic Software Product Lines (DSPL) with runtime sharing and variation among tenants. We first identify the different kinds of changes to a service-based SaaS application, and the consequential impacts of those changes. We then discuss how to realize and manage each change and its resultant impacts in the DSPL. A software engineer declaratively specifies changes in a script, and realizes the changes to the runtime model of the DSPL using the script. We demonstrate the feasibility of our approach with a case study.

Keywords: SaaS, Evolution, Multi-tenancy, SPL, Compositional, Feature.

1 Introduction

The Software as a Service (SaaS) models for service delivery offer software applications as a utility over the Internet. In particular, the Single-Instance Multi-Tenancy (SIMT) model hosts different tenants in a single application instance, increasing runtime sharing and hence reducing operational cost [1]. In this model, the functionality for all the tenants is integrated into a single application, and the differentiation of the varied support for tenants is realized at runtime.

After an SIMT application is successfully developed and deployed, its evolution takes place. During this phase, the application can be modified, for instance, to cope with a changed need of a tenant or the SaaS provider or a change in a partner service's capability. Evolving an SIMT application is a complex problem. Firstly, the application should support *different classes of changes* that can potentially occur during its lifetime. Secondly, the application needs to enable the identification and control of the *impacts of a change* on the application. Finally, a change and its impacts need to be *realized and managed at runtime* without disturbing the operations of those tenants unaffected by the change.

To date, there is little support for runtime evolution of a multi-tenant SaaS application [2-4]. Some efforts have considered such issues as tenant on-boarding [2] and tenant-specific variations [3, 4]. However, they do not sufficiently support two key activities of change management [5]: identifying a change and its impacts, and designing and implementing the change and impacts.

In [6], we have proposed to realize a service-based SIMT SaaS application as a Service-Oriented Dynamic Software Product Line (SO-DSPL) that supports runtime sharing and variation across tenants/products. Our approach utilizes the DSPL's capability to share and differentiate product features, but all achieved at *runtime*.

In this paper, we discuss the above-mentioned *two key activities of change management* (main contributions) for service-based SIMT applications developed using our product line based model. We first identify the different types of changes to our SO-DSPL and their potential impacts. Second, we discuss our approach to realizing each change and managing each change impact. In particular, we support the identification of the potential impacts of a change on the products (tenants), and the management of such impacts without disturbing the operations of the unaffected products. An initial modification and its consequential impacts can be specified and realized through the runtime representation of the product line created based on the *models@runtime* concept [7]. With a case study that implements common SPL evolution scenarios, we demonstrate the feasibility of our approach. We analyze the case study results to assess change impacts and the programming effort for the scripts that specify changes. We also quantify the time taken to realize such changes at runtime.

In this paper, we start by providing the motivation, background, and overview of our approach to realizing an SIMT application as an SO-DSPL (Sections 2, 3, and 4). Section 5 presents our approach to the identification and management of the runtime changes and their impacts. In Sections 6 and 7, we discuss our prototype implementation and evaluation results respectively. Section 8 presents related work, and Section 9 concludes the paper while providing directions for future research.

2 Motivating Scenario and General Requirements

To motivate this research, let us consider a business scenario from SwinRoadside, a company offering roadside assistance to its customers such as Travel Agencies (TA) and Truck Sellers (TS) by utilizing external partner services such as Garage Chains (GC) and Towing Companies (TC). SwinRoadside manages both the roadside assistance business and the supporting IT infrastructure, which adopts the *SIMT SaaS model*. The customers use their own variants of this roadside assistance service to serve their users such as travelers and truck buyers. In this IT-enabled business scenario, we can identify three key requirements for SwinRoadside.

(Req1) *Runtime Sharing with Variations.* To achieve economies of scale, SwinRoadside expects to share the roadside business process and services across its customers (tenants). However, these customers have varying needs. For instance, TA1 needs onsite vehicle repair and accommodation, while TS1 prefers repairing at a garage. SwinRoadside needs to support sharing with variations at runtime.

***(Req2)** Managing Runtime Changes to the SaaS Application.* The requirements of the tenants, the SaaS provider, and the external services can change over time. For instance, six months into operation, TA1 needs support for renting a vehicle instead of accommodation, as travelers prefer continuing their journey. After one year, SwinRoadside decides to enhance repair notification by supporting the direct notification of a motorist by the garage. The towing company starts to offer accident towing that SwinRoadside and some of its customers want to utilize. The roadside application needs to evolve at runtime to respond to or utilize these changes.

***(Req3)** Managing Change Impacts.* A change in the roadside process can affect the application as well as individual tenants. For example, modifying the towing capability to tow a rented vehicle (for TA1) can affect TS1 that uses it to tow a vehicle to a garage. The roadside application needs to identify and control these impacts.

3 Software Product Lines and Feature Model

An SPL is a family of software systems developed from a common set of core assets [8]. Compared to an SPL, a dynamic SPL (DSPL) creates and adapts products at runtime [9]. The realization of a variant-rich application with the SPL approach can yield significant improvements in business drivers such as time to market, cost, and productivity. There are two main ways to implement an SPL: *annotative approach* and *compositional approach* [10]. The former embeds the features of the SPL in the application code using explicit annotations (e.g., '#ifdef' and '#endif' statements of C style) or implicit annotations (e.g., hooks or extension points), supporting fine-grained variability, but reducing flexibility and maintainability [10]. The latter realizes the features as modular units and creates products by composing these modular units. It can potentially reduce the aforementioned drawbacks of the annotative approach [10].

A feature model [11] captures the commonality and variability in a product line at a higher abstraction level. It supports activities such as asset development and product creation. Figure 1 shows the cardinality-based feature model [12] for the motivating example. The *Composed of* relationship arranges features in a *hierarchy*. For instance, the features *Accommodation* and *TechAdvice* are the children of *ExtraServices*. The feature cardinality specifies how many instances of a feature can be included in a feature configuration or product. The cardinality of an *optional* feature is [0-1], and that of a *mandatory* feature is [1-1]. The cardinality of a feature group specifies how many features the group can include. For example, the group cardinality ([1-2]) of *ExtraServices* implies that *at least one* of its two children must be selected. The constraints define dependencies among features. For instance, the constraint *AtGarage includes Tow* indicates that if *AtGarage* is selected, *Tow* must also be selected. By selecting the features respecting these constraints, a feature configuration is created.

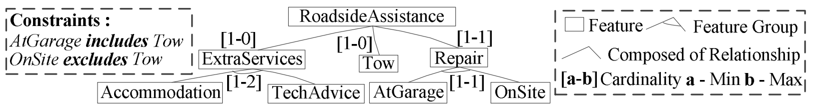


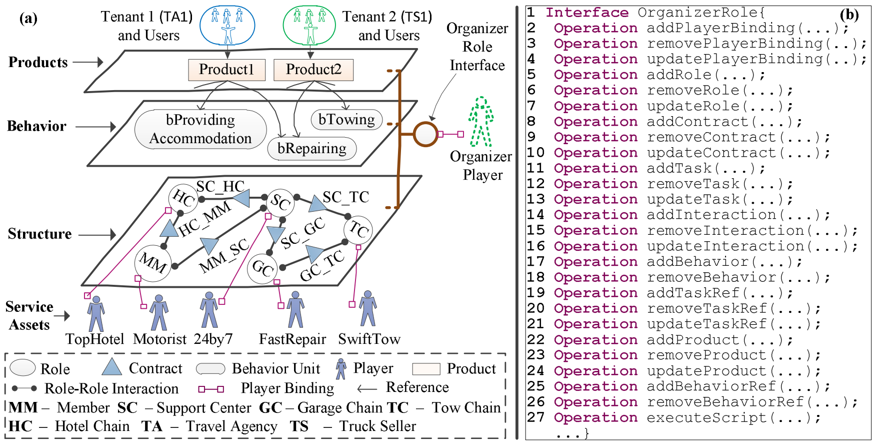
Fig. 1. The feature model for the motivating example

4 Product Line-Based Realization of SIMT SaaS Applications

Design-Time Representation. In [6], we have introduced a service-oriented DSPL (SO-DSPL) based approach to realizing service-based SIMT SaaS applications that supports runtime sharing among products/tenants while allowing product/tenant-specific variations (***Req1***). This section provides an overview of our DSPL based approach, which comprises four layers, as shown in Fig. 2(a).

At the bottom is the *service asset layer*, including all the partner services used by the product line or SaaS application. The *structure layer* provides an abstraction over *service assets* and their interactions needed to realize the *features* of the product line. The *roles* are abstract representations of service assets (referred as *players*), making roles and players loosely coupled. The *contracts* capture the allowed interactions between the players playing roles, and make roles loosely coupled. The *role-contract topology*, consisting of roles and contracts, models the structure of the product line. A role defines its *responsibility* as a set of *tasks* that encapsulate the *capabilities* of a service. A contract consists of a set of *interaction terms*, defining the allowed interactions between the relevant roles (players). The input and output of a task are defined based on interaction terms. Messages flow between services via roles and contracts.

Consider the structure layer for the motivating example (see Fig. 2(a)). The role-contract topology comprises the roles MM, SC, HC, GC, TC and the contracts among them. The role GC, an abstract garage service, is realized by the service (player) FastRepair. The role SC represents the Support Center and is realized by the service 24by7. The contract SC_GC defines the expected interactions between the players 24by7 and FastRepair in playing the roles SC and GC. Lines 13-14 in Fig. 3 show the task *tRepair* of the role GC. The task's input (*UsingMsg*) uses the interaction *iOrderRepair* from SC. Its output (*ResultingMsg*) refers to the interactions *iPayRepair* to SC and *iAckRepair* to *MM* (representing the member, i.e., the user or motorist).



```
Interface OrganizerRole{
 Operation addPlayerBinding(...);
 Operation removePlayerBinding(..);
 Operation updatePlayerBinding(..);
 Operation addRole(...);
 Operation removeRole(...);
 Operation updateRole(...);
 Operation addContract(...);
 Operation removeContract(...);
 Operation updateContract(...);
 Operation addTask(...);
 Operation removeTask(...);
 Operation updateTask(...);
 Operation addInteraction(...);
 Operation removeInteraction(...);
 Operation updateInteraction(...);
 Operation addBehavior(...);
 Operation removeBehavior(...);
 Operation addTaskRef(...);
 Operation removeTaskRef(...);
 Operation updateTaskRef(...);
 Operation addProduct(...);
 Operation removeProduct(...);
 Operation updateProduct(...);
 Operation addBehaviorRef(...);
 Operation removeBehaviorRef(...);
 Operation executeScript(...);
  ...}
```

Fig. 2. (a) An overview of the SO-DSPL for the motivating example, (b) part of the organizer

```
1  ProductDefinition Product1 {
2    CoS "eProduct1Reqd"; CoT "ePaidRepair * ePaidRoomRent * eAckedMM";
3    BehaviorRef  bRepairing; BehaviorRef  bProvidingAccommodation; ... }
4  Behavior bRepairing {
5   TaskRef GC.tRepair     { InitOn "eRepairReqd"; Triggers "eRepaired"; }
6   TaskRef SC.tPayRepair { InitOn "eRepaired"; Triggers "ePaidRepair"; }
7   TaskRef MM.tAckRepair { InitOn "eRepaired"; Triggers "eAckedMM";     } ...}
8  Behavior bProvidingAccommodation { ... }
9  Contract SC_GC { A is SC, B is GC;
10   ITerm iOrderRepair(String:msg) withResponse (String:ack)from AtoB; ITerm iPayRepair(..);..}
11 Contract GC_MM { ITerm iAckRepair(...) ...}
12 Role GC {
13  Task tRepair { UsingMsgs SC_GC.iOrderRepair.Req;
14                 ResultingMsgs SC_GC.iPayRepair.Req,GC_MM.iAckRepair.Req; } ...}
15 Role MM { Task tAckRepair{...} ...}, Role SC { Task tPayRepair{...} ... }
16 PlayerBinding gcPlayer "www.fastrepair.com/GCService" is a GC;
```

Fig. 3. Part of the configuration for the product line depicted in Fig. 2(a)

The *behavior layer*, consisting of *behavior units*, encapsulates the control flow and regulates the message flow between service assets. To provide a *feature*, a behavior unit realizes a collaboration among a subset of services by coordinating the tasks of the roles that these services fulfill. The topology of the collaboration (referred to as *local topology*) is defined using references to the tasks of the participating roles. The *control flow* is specified as the dependencies between the tasks using their *InitOn* and *Triggers* clauses (pre- and post-conditions) based on events that are generated by interpreting role-role interactions. For example, consider the behavior unit *bRepairing* (lines 4-7 in Fig. 3). It groups and coordinates the tasks of *GC.tRepair*, *SC.tPayRepair*, and *MM.tAckRepair*. The task *GC.tRepair* depends on the task that creates the event *eRepairReqd*. Its completion generates the event *eRepaired* that triggers (as the preconditions for) the consequent tasks, e.g., *SC.tPayRepair* and *MM.tAckRepair*.

At the *product layer*, a product models a tenant's product configuration and composes the related behavior units by referring to them (*aka*, the *compositional approach*). Products *share* behavior units for their commonality and use different behavior units for their variability, i.e., achieving the *SIMT model*. As depicted in Fig. 2(a), Product1 and Product2 use the behavior unit *bRepairing*, and one of the behavior units *bTowing* and *bProvidingAccommodation*. A product also defines its start and end using CoS (Condition of Start) and CoT (Condition of Termination) (line 2 in Fig. 3).

Runtime Representation. The above-mentioned architecture model of the product line is kept alive at runtime using the *models@runtime* concept [7]. As such, its elements can be modified at runtime, e.g., adding roles or contracts. In particular, it has an organizer role and player (see Fig. 2(a)) through which runtime changes to the product line can be performed (see Section 5). The organizer role and player are generic to our approach. Some of their adaptation capabilities that this research uses are shown as adaptation operations in Fig. 2(b).

5 Runtime Evolution of Product Line-Based SIMT SaaS Applications

Two of the main activities for software change management are: (1) identifying a change and its impacts, and (2) designing and implementing the change [5, 13]. In this paper, we consider these activities *at runtime* for service-based SIMT SaaS applications developed using our DSPL based model (***Req2*** and ***Req3***). Section 5.1 identifies the *changes* to the DSPL and their *potential direct impacts*. Section 5.2 discusses the realization and management of the identified changes and impacts in the DSPL.

5.1 Identification of Changes and Impacts

A change request and the current system are key inputs to a change process [5]. In our approach, external service providers consider changes at the service-level, such as service addition and removal. The SaaS provider and tenants identify changes at the feature-level as addition, removal and modification of features. The SaaS provider can also consider architectural changes, e.g., for the purpose of optimizing the product line architecture. In general, each layer of the product line can be potentially modified to realize a change (see Fig. 4).

Service Asset Layer. The changes at the service asset layer include: adding, removing, replacing, and modifying a service asset, service capability changes, and service interface changes. The capability changes include adding, removing, and modifying capabilities and the control relations between them. The interface changes include adding, removing, and modifying operations and the control relations between them.

A new service asset (to be used) requires a role, a player binding for that role, and a set of contracts to capture the expected relationships between the new service asset and the relevant existing service assets in realizing that role. It also introduces new capabilities. The removal of a used service asset makes the related player binding, role, and contracts invalid since the player binding refers to a nonexistent service, the realization of the role is removed, and the contracts represent nonexistent relationships. Moreover, the used capabilities of the deleted service asset are removed. The replacement of a used or an existing service asset requires updating the related player binding. Additionally, the mismatch/difference between the new service and the replaced one can result in capability and interface changes (see below for their impacts). A modification to a used service asset can involve capability and interface changes.

A new service capability (to be used) requires a task to represent it. The removal of a used capability makes the related task invalid since there is no realization for it. The modifications to used capabilities (e.g., merging capabilities) can result in the same types of changes to the relevant tasks. Generally, to use or realize a capability, service

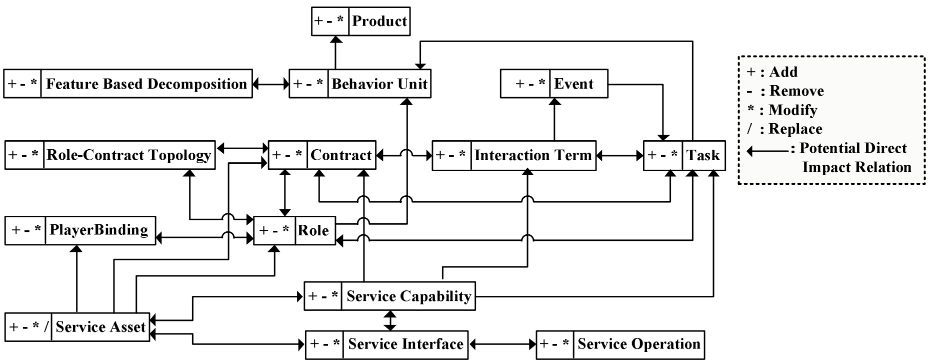


Fig. 4. Changeable elements and their *potential direct impact* relations

assets need to interact with each other, and thus a capability change can also have impacts on contracts and interaction terms. A change to a control relation between used capabilities can affect the dependencies among the corresponding tasks captured in relevant behavior units. A capability change can also involve an interface change.

An interface change related to an unused capability does not affect the product line. The impacts of an interface change related to a used capability (unchanged) are unwanted by the product line, and thus should be controlled (see Section 5.2). An interface change that alters a used capability has the same impacts of a capability change.

Structure Layer. The changes at the structure layer include adding, removing, and modifying the role-contract topology and the player bindings. The modifications to a role-contract topology include adding, removing, and modifying roles and contracts. Altering a role involves adding, removing, and modifying tasks. Altering a contract involves adding, removing, and modifying interaction terms. The modifications to a player binding include updating its endpoint and role reference.

The addition and removal of the role-contract topology implies the initiation and termination of the system. A new role may need a set of tasks, a player binding, and the contracts with the other roles that the new role should interact. The removal of a role deletes its tasks, and makes its contracts and the references to the role in behavior units and player bindings invalid. A new contract between two roles relates the two roles, and may require a set of interaction terms to be used by the tasks of the two roles. A deletion of a contract removes its interaction terms and the association between the related roles, and makes the references to it by the related tasks invalid.

A new task may use a subset of existing interaction terms, and require the references to it in the behavior units that need to use it. If an interaction term used in a task is not shared by other tasks, the removal of the task makes the interaction term isolated. A deleted task also makes the reference to it in the related behavior units invalid.

A new interaction term may require adding the references to it in the tasks that need to use it. A deleted interaction term makes the references to it in the related tasks invalid. Moreover, the changes to interaction terms that alter the events they create can affect the representations of the dependencies between tasks (*InitOn/Triggers* of a task reference). A new incoming interaction of a task may require adding the relevant events to the *InitOn* of the related task references. A removed incoming interaction makes the references to the related events invalid. The similar impacts on the *Triggers* of a task reference can occur due to a change to an outgoing interaction of a task.

A new player binding for a role makes the role implemented by a player. A deleted player binding removes the realization for the corresponding role.

Behavior Layer. The changes at the behavior layer include creating, deleting, and modifying the *feature-based decomposition* of the behavior layer. The alterations to this decomposition involve adding, removing, and modifying behavior units. The modifications to a behavior unit involve adding, removing, and modifying its task references. The alterations to a task reference include adding, removing, and updating its *InitOn* and *Triggers* clauses to create and change the dependencies between tasks.

The creation and deletion of the feature-based decomposition of the behavior layer implies the initiation and termination of the system. A new behavior unit requires the references to it in the products that need to use it. A deleted behavior unit makes the references to it in the relevant products invalid. The changes to the task references and task dependencies captured in a behavior unit can alter the service collaboration (the feature implementation) realized by the behavior unit. This in turn can modify the feature (an end-user experienced functionality/behavior) offered by the behavior unit.

A change to a feature implementation can introduce unintended behaviors to one or more different features as well as to a subset of the products that use the feature. As an example for the first case, consider that the feature *AtGarage* uses the feature *Tow* to carry a vehicle to a garage (used by Product2), and the new feature *VehicleHire* also needs the feature *Tow* to tow a rented vehicle (to be used by Product1). Changing the collaboration related to the feature *Tow* can affect the feature *AtGarage.* As an example for the second case, suppose that Product2 needs a periodic repair notification. Modifying the collaboration for the shared feature *Repair* for this purpose adds an unwanted behavior to Product1. These effects need to be reduced (see Section 5.2).

Product Layer. The changes at the product layer include: adding, removing, and modifying products. The modifications to a product involve adding and removing the references to behavior units, and updating its CoT and CoS. Since the events used in the CoT and CoS of a product depend on the behavior units that the product uses, the inclusion and exclusion of a behavior unit in the product as well as the change to a behavior unit used by the product can affect the CoT and CoS of the product.

5.2 Realization of Changes and Impacts

In this section, we describe how the identified changes and impacts can be realized in the SO-DSPL, and how the change impacts are managed and realized.

Solutions for Changes. The change primitives supported by the organizer (see Fig. 2 (b)) are used to perform the changes to the runtime model of the product line.

Using the operations *[add/remove/update][Role/Contract]()*, the role-contract topology can be altered. The methods *[add/remove/update]PlayerBinding()* can be used to realize player binding changes. To change tasks, interaction terms, and their relations, the operations *[add/remove/update][Task/Interaction]()* can be used.

The operations *[add/remove]Behavior()* need to be used to add or remove a behavior unit. By changing the task references of a behavior unit using the methods *[add/ remove]TaskRef()*, the local topology of a collaboration captured in a behavior unit can be modified. The control flow can be altered by modifying dependencies among tasks via changing their *InitOn* and *Triggers* using the operation *updateTaskRef()*. For example, to ensure a repair notification (*MM.tAckRepair*) follows a repair payment (*SC.tPayRepair*), the *InitOn* of the taskref *MM.tAckRepair* in the behavior unit *bRepairing* (Fig. 3) can be replaced by the *Triggers* of *SC.tPayRepair*. Figure 5 shows this variation with an EPC (event-driven-process chain) diagram [14].

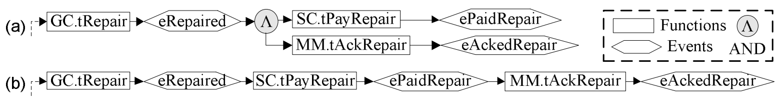


Fig. 5. Changing the control flow via altering task dependences (a) initial, (b) modified

A product can be created and removed using the methods *[add/remove]Product()*. The operation *updateProduct()* can be used to alter the CoS and CoT of a product. A created product can be reconfigured using the operations *[add/remove]BehaviorRef()*.

Solutions for Impacts. The general approach to realize an impact of a change is as follows. If a change E causes a change F as a direct impact, then to propagate this impact, the techniques for realizing the change F need to be used (see above). For example, the removal of a role requires the removal of its contracts since there are invalid, and the operation *removeContract()* can be used to propagate this effect. Note that we assume that the initial change made by a developer is an intended one. Due to limited space, we do not describe each propagation link using the general approach.

However, there are two cases that require specific techniques to control the propagation of impacts. First, the service interface changes related to a used capability (unchanged) should not be propagated to the product line. Such changes include: operation signature changes, and operation granularity (e.g., split) and transition (control relation) changes. By changing the transformations between role-role interactions and role-player interactions, the propagation of the operation signature changes can be avoided. To cope with the operation granularity and transition changes, sub-service composites that act as *adapters* need to be created. A sub-composite (an adapter) for handling the interface changes of the player C of the role B becomes the new player (the realization) for the role B. In [15], different service composite-based adapters are presented, and thus we will not further discuss these issues in this paper.

Second, the feature changes that add unwanted behaviors to one or more different features or to a subset of products need to be controlled. For this purpose, we create *variations* of the affected feature implementations (collaborations). Such variations are captured in the behavior layer by *specializing* the related behavior units. Note that a variation in a collaboration may require structural changes, e.g., new tasks. A behavior unit can be specialized to create new child behavior units by adding new elements or overriding its existing elements. The parent represents common behaviors, and the children represent variations. For instance, to support towing a rented vehicle, the behavior unit *bTowing* can be extended to create *bTowingRentedVehicle* (Fig. 6). The task *VC.tGetLocation* is created (VC - vehicle renting company), and a reference to it is added to *bTowingRentedVehicle*. The *InitOn* of *TC.tTow* is overridden to ensure that the towing starts after *VC.tGetLocation* gave the destination. Now, Product1 uses *bTowingRentedVehicle*, while Product2 continues using *bTowing* (no impact).

Note that, due to limited space, the use of the proposed techniques to solve feature implementation dependency types that can make products invalid, e.g., operational dependency [16] and feature interaction [17] is not discussed. To address these issues, the works in [16, 17] used (class) inheritance and coordination, which we also adopt.

```
Behavior bTowing {
 TaskRef TC.tTow { InitOn "eGCLocKnown"; Triggers "eTowed"; } ...  }
Behavior bTowingRentedVehicle extends bTowing {
 TaskRef VC.tGetLocation { InitOn  "eTowReqd"; Triggers "eVCLocKnown"; }
 TaskRef TC.tTow { InitOn "eVCLocKnown"; Triggers "eTowed"; } ...  }
```

Fig. 6. Extending the collaboration for the feature *Tow* for the feature *VehicleHire*

Realization of Changes and Impacts to the Running Application. Upon receiving a change request, the software engineer identifies the initial changes to realize the change request as well as the further impacts of each change. The solutions for the identified changes and impacts are designed and then specified in a form of a change script. A unit in such a script is a *change command*, which comprises a name and a set of parameters as name-value pairs. For example, *addBehavior* is a command name, and *bId ="bRentingVehicle"* is a parameter (line 18 in Fig. 7). These change commands are the representations of the change primitives of the organizer at the script-level. The changes defined in a change script can be applied to the running system using the operation *executeScript()* of the organizer role. The organizer creates the executable change commands (in Java) from a change script, and applies those commands to the runtime model of the system created using the *models@runtime* concept.

An Example. Bellow, we present the process of designing a change script using an example: *add feature VehicleHire* whose implementation creates a new collaboration among a subset of services to implement the feature *VehicleHire* for use in Product1.

1. *Identifying and defining role-contract topology and service changes*: A developer identifies the differences between the expected topology and the initial one, and specifies the differences in a change script. For instance, the collaboration for *VehicleHire* requires a topology consisting of the roles MM, VC (vehicle renting company) and SC, and contracts SC_MM, SC_VC and VC_MM. The roles MM and SC, and contract SC_MM are in the initial system so the required changes concern the role VC, and contracts SC_VC and VC_MM. The player *TomAuto* is required to play role VC. Lines 3, 8, and 15 in Fig. 7 define part of these changes.
2. *Identifying and defining role-role interaction changes*: Next, a developer designs the changes to interaction terms. In our example, we add the interaction terms *iOrderVehicle* and *iPayVehicleFare* to the contract SC_VC, and the interaction term *iAckVehicleBooking* to the contract VC_MM. Lines 9-10 and 12 in Fig. 7 specify part of these changes.

rId : role id	**cId** : contract id	**bId** : behavior id
tId : task id	**tmId** : term id	**pdId** : product id

```
1  AddFeatureVehicleHire{
2   // Roles related changes
3   addRole rId="VC";
4   addTask rId="VC" tId="tRentVehicle" usingMsgs="SC_VC.iOrderVehicle.Req"
5    resultingMsgs="SC_VC.iOrderVehicle.Res, VC_MM.iAckVehicleBooking.Req";
6   ...
7   // Contracts related changes
8   addContract cId="SC_VC" rAId="SC" rBId="VC";
9    addInteraction tmId="iOrderVehicle" cId="SC_VC" direction="AtoB" ...;
10   addInteraction tmId="iPayVehicleFare" cId="SC_VC" direction="BtoA" ... ;
11  addContract cId="VC_MM" rAId="VC" rBId="MM";
12   addInteraction tmId="iAckVehicleBooking" cId="VC_MM" direction="AtoB" ...;
13  ...
14  // Player-bindings related changes
15  addPlayerBinding pbId="vcPb" rId="VC" endpoint="www.tomauto.com/VCService";
16  ...
17  // Behavior units related changes
18  addBehavior bId="bRentingVehicle";
19  addTaskRef tId="VC.tRentVehicle" bId="bRentingVehicle"
20       initOn="eRentVehicleReqd" triggers="eVehicleRented";
21  ...
22  // Products related changes
23  addBehaviorRef pdId="Product1" bId="bRentingVehicle";  ... }
```

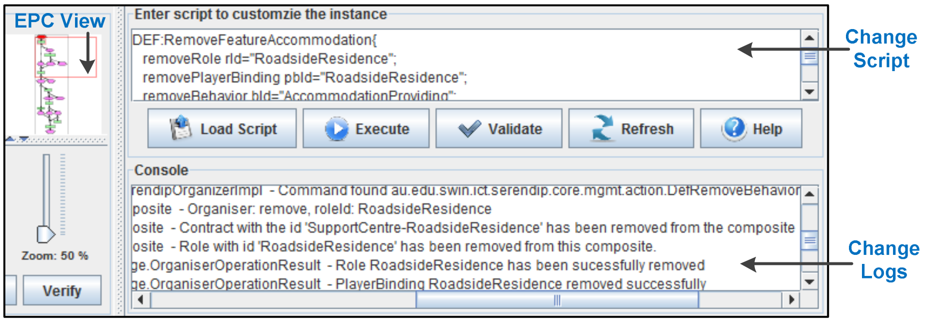
Fig. 7. Part of the change script for adding the feature *VehicleHire*

3. *Identifying and defining task definition changes*: Next, a developer identifies and designs the changes to the task definitions in the related roles. In our example, we create the definitions for the tasks *VC.tRentVehicle, SC.tPayVehicleFare,* and *MM.tAckVehicleBooking*. Lines 4-5 in Fig. 7 define part of the task *tRentVehicle*.
4. *Identifying and defining behavior unit changes*: In the next step, the changes to the local topologies, control flow, and behavior layer decomposition are designed. In our example, the behavior unit *bRentingVehicle* needs to be created with the references to the above-mentioned tasks. Lines 18-20 specify part of these changes.
5. *Identifying and defining product changes*: Next, a developer reconfigures the affected products. In our example, Product1 needs the feature *VehicleHire*. Thus, a reference to the behavior unit *bRentingVehicle* is added to it (line 23 in Fig. 7).

6 Prototype Implementation

To realize SO-DSPL based SaaS applications in our approach, we adopt and further improve the ROAD/Serendip framework [18, 19], which supports development and management of adaptive service orchestrations. In doing so, we treat the SO-DSPL realization for an SIMT SaaS application as an adaptive service orchestration. We presented this implementation in [6] in detail. For this work, we have used this prototype to analyze the changes and impacts presented in Section 5 and to implement the proposed solutions. We have also formulated and implemented the change commands required for this work, which are generic to our approach and independent from a particular case study.

We provide Eclipse plugins to specify (the change script editor) and perform (the adaptation tool) changes discussed in Section 5. The former can highlight and detect errors of the syntax of change commands. The latter allows executing a change script, shows the status of the execution, and if it fails, the details required to correct and rerun it. The organizer role is exposed as a Web service to allow providing change scripts remotely. We adopt the Serendip language to specify the SO-DSPL architecture and the evolutionary changes. Figure 8 shows a screenshot of the adaptation tool, executing the change script for removing the feature *Accommodation*. The snippets of the change script and the logs of the execution of the script are shown.



```
DEF:RemoveFeatureAccommodation{
  removeRole rId="RoadsideResidence";
  removePlayerBinding pbId="RoadsideResidence";
  removeBehavior bId="AccommodationProviding";
```

```
rendipOrganizerImpl  - Command found au.edu.swin.ict.serendip.core.mgmt.action.DefRemoveBehavior
posite  - Organiser: remove, roleId: RoadsideResidence
osite  - Contract with the id 'SupportCentre-RoadsideResidence' has been removed from the composite
osite  - Role with id 'RoadsideResidence' has been removed from this composite.
ge.OrganiserOperationResult  - Role RoadsideResidence has been sucessfully removed
ge.OrganiserOperationResult  - PlayerBinding RoadsideResidence removed successfully
```

Fig. 8. A screenshot of the adaptation tool running the script for removing *Accommodation*

7 Evaluation

We demonstrate our approach's feasibility by realizing 10 SPL evolution scenarios (adapted based on [20, 21]) (Table 1). For each scenario, we create a change script capturing the difference between the initial system configuration and the expected configuration after an evolution. A change script is enacted at runtime on the system with the initial configuration. To validate an evolution, we first analyze logs for all the expected changes. Second, we compare the responses and logs for requests to the products after evolution with those of the system having the expected system configuration (manually created). The case study resources are in http://tinyurl.com/d5xlaom.

Change Impact Analysis. We assessed the effectiveness of our support for evolution by doing a change impact analysis. The complexity of each scenario was intentionally kept low to make it easier to identify change impacts. Due to limited space, we present the results for three scenarios related to addition, removal, and modification of a feature. The results for other scenarios are in the case study resources (see above).

To add the feature *BankTransfer* (CS3), we create a collaboration consists of the roles BK (Bank), AF (AccountingFirm), and MM. The last two roles and the contract between them (AF_MM) are part of the initial system. Two new contracts BK_AF and BK_MM are created. The tasks and interactions required to implement the bank transfer functionality are added/modified. A new player for realizing the role BK, and the player-binding is added. Finally, the behavior unit *bPayingByBankTransfer,* to capture this collaboration is created, and the related products are updated to use it.

The removal of the feature *BankTransfer* (CS6) is realized by deleting the elements of the collaboration for that feature, which are the same elements introduced in CS3.

Scenario CS10 is implemented by removing the interaction term *iNotifyCompletion* from the contract SC_MM, and adding a new contract GC_MM with the same interaction term. Additionally, the tasks *GC.tOrderRepair* and *SC.tAckRepair* are updated to reflect the interaction term changes, and the new task *MM.AckRepair* is added. These modifications are confined to the collaboration for the feature *Repair*.

As per the above analysis, units of change at the feature-level and the service-level are confined to their explicit representations in the SO-DSPL architecture, i.e., collaborations and abstract representations of services, their interactions and the control flow among them. This is a key requirement to support effective evolution [22].

Table 1. Change scenarios for the roadside assistance case study

No:	Type of Change	Example
CS1	Inclusion of a mandatory feature	Supporting the reimbursement of costs met by a member
CS2	Inclusion of an optional feature	Supporting renting a vehicle as an alternative transport
CS3	Inclusion of an alternative feature	Supporting paying by credit card or bank transfer
CS4	Mandatory to optional conversion	Allow using towing or expert advice without repairing
CS5	Removal of an optional feature	Discontinuing providing accommodations
CS6	Removal of an alternative feature	Dropping the bank transfer payment option
CS7	Splitting one feature into two	Separating legal assistance from the accident towing
CS8	Merging two features	Merging technical advice and vehicle test reports
CS9	Feature implementation changes	Extending fuel delivery by using an external service center
CS10	Feature implementation changes	Direct notification by a garage instead via a support center

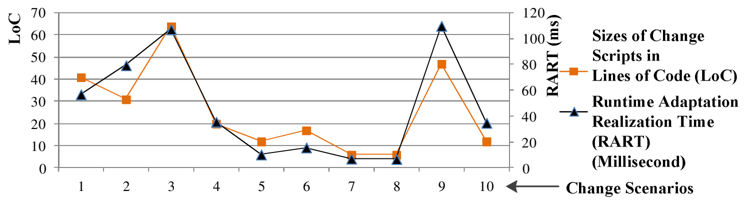


Fig. 9. Runtime adaptation realization time and change script size for change scenarios

Programmer Effort. We have used lines of code (LOC) as the metric to measure the size of a change script to approximate the effort for developing the change scripts for the case study (similar to [23]). We ignored blank lines and comments. The length of a line is approximately 120 characters. Figure 9 shows the sizes of the change scripts.

Runtime Adaptation Realization Time (RART). We have measured the *runtime adaptation realization time (RART)* for each scenario. It is the time difference between the system receiving a script and its being ready for use after changes. The framework was run on an Intel i5-2400 CPU, 3.10GHz with 3.23 GB of RAM and Windows XP. As shown in Fig. 9, the RART is within 6-110 milliseconds. We believe that this is reasonable. In addition, we observed a correlation between the RART and the size (LoC) of a change script, which approximately corresponds to the number of atomic adaptation steps included in the script. We also observed that the time taken for the removal of a feature (CS3) is low compared to its addition (CS6).

8 Related Work

We discuss below the existing research efforts from the perspectives of (D)SPLs and SaaS applications that consider *service-based systems* and support *runtime* changes.

In general, the *runtime* changes to a product line fall into two categories: adaptation of a product, and evolution of the product line [9]. Most existing works studied only the first issue [9]. We also considered it in [6] and thus focus on the second issue in this paper. In a product line, the problem space (e.g., the feature model), the solution space, and the mapping between them can evolve [20]. Within the scope of this paper, we consider the solution space for an SO-DSPL. Among the works focused on the solution space, Morin et al. [7] and Baresi et al. [21] supported modifying a business process at a set of *predefined* variation points to create products. They used SCA (Service Component Architecture) and BPEL (Business Process Execution Language) to realize their SO-DSPLs, and AOP (Aspect-Oriented Programming) to realize changes. Bosch and Capilla [24] supported, in a smart home SPL, feature-level changes by mapping a feature to a device that offers a particular service.

Studies on runtime changes to SaaS applications considered issues such as tenant on-boarding [2] and tenant-specific variants [3, 4]. Ju et al. [2] proposed a formal model to assess the cost of tenant on-boarding. In the context of *component-based systems*, Truyen et al. [3] proposed the tenant-aware dependency injection to bind tenant-specific variants to the variation points of the application' component model. Moens et al. [4] proposed a feature-model based development of services where a one-to-one mapping between a feature and a service is used. These services are deployed in a cloud environment and composed based on the selected features.

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

Criterion \ Approach		[7]	[21]	[24]	[3]	[4]	we
Req1	Runtime Sharing	-	-	-	+	-	+
	Variations	+	+	+	+	+	+
Req2 : Managing Changes		~	~	~	-	-	+
Req3 : Managing Change Impacts		-	-	-	-	-	+
Explicit Representations of Units of Change		~	~	-	-	-	+

+ Supported
- Not Supported
~ Limited Support

In analyzing the above works, the studies that allow modifying the product line or SaaS application at the predefined variation points did not consider the changes to the base model and its variation points, and the studies that used a compositional approach assumed a feature as a component service. None of them considers change impacts on variants. The underlying technologies used (i.e., SCA and BPEL) do not sufficiently represent the structure and behavior of services or modular service collaborations, and thus offer little support to explicitly represent units of change at the feature-level or the service-level. Moreover, the works in DSPLs usually create physically separated variants, which do not meet the requirements of the SIMT model.

In comparison with the above approaches (Table 2), we use a compositional technique to realize the variability by treating collaboration as the unit of composition. A collaboration provides a better abstraction to modularize a feature compared to a service or an arbitrary process fragment [6]. Moreover, we consider the runtime changes to an SO-DSPL that supports runtime sharing, and the management of the impacts of those changes. Our product line architecture provides an abstraction over the service asset space and explicitly represents features as modular units.

9 Conclusions and Future Work

We have addressed the runtime evolution of single-instance multi-tenant SaaS applications that are realized based on SO-DSPLs and support runtime sharing and tenant-specific variations. We have identified different types of changes to the SaaS application and their potential impacts, and proposed techniques to realize those changes and impacts at our SO-DSPL based SaaS applications. In particular, we have presented solutions to control the impacts of a change on tenants (products). A change is realized on the runtime model of the product line created based on the models@runtime concept. We have evaluated our approach with a case study and related analysis concerning change impacts, effort of developing change scripts, and time to realize a runtime change. The results have shown that our approach is feasible and beneficial.

In future, we plan to extend FeatureIDE (http://fosd.de/fide) to provide direct support for feature-based evolution, to study change impacts on ongoing transactions, and to explore the performance variability in a service-based SaaS application.

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

References

1. Chong, F., Carraro, G.: Architecture Strategies for Catching the Long Tail, MSDN Library. Microsoft Corporation (2006)
2. Ju, L., Sengupta, B., Roychoudhury, A.: Tenant Onboarding in Evolving Multi-tenant Software-as-a-Service Systems. In: ICWS, pp. 415–422 (2012)
3. Truyen, E., et al.: Context-oriented programming for customizable SaaS applications. In: SAC, pp. 418–425 (2012)
4. Moens, H., et al.: Developing and managing customizable Software as a Service using feature model conversion. In: NOMS, pp. 1295–1302 (2012)
5. Bohner, S.A.: Impact analysis in the software change process: a year 2000 perspective. In: ICSM, pp. 42–51 (1996)
6. Kumara, I., et al.: Sharing with a Difference: Realizing Service-based SaaS Applications with Runtime Sharing and Variation in Dynamic Software Product Lines. In: SCC, pp. 567–574 (2013)
7. Morin, B., et al.: Models@ Runtime to Support Dynamic Adaptation. Computer 42, 44–51 (2009)
8. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Wesley (2003)
9. Bencomo, N., Hallsteinsen, S., Almeida, E.S.: A View of the Dynamic Software Product Line Landscape. Computer 45, 36–41 (2012)
10. Kastner, C., Apel, S., Kuhlemann, M.: Granularity in Software Product Lines. In: ICSE, pp. 311–320 (2008)
11. Kang, K.C., Lee, J., Donohoe, P.: Feature-oriented product line engineering. IEEE Software 19, 58–65 (2002)
12. Czarnecki, K., Kim, C.H.P.: Cardinality-based feature modeling and constraints: a progress report. In: International Workshop on Software Factories, pp. 16–20 (2005)
13. Han, J.: Supporting impact analysis and change propagation in software engineering environments. In: STEP, pp. 172–182 (1997)
14. Scheer, A.W., Thomas, O., Adam, O.: Process Modeling using Event-Driven Process Chains. In: Process-Aware Information Systems, pp. 119–145 (2005)
15. Benatallah, B., Casati, F., Grigori, D., Nezhad, H.R.M., Toumani, F.: Developing Adapters for Web Services Integration. In: Pastor, Ó., Falcão e Cunha, J. (eds.) CAiSE 2005. LNCS, vol. 3520, pp. 415–429. Springer, Heidelberg (2005)
16. Lee, K., Kang, K.C.: Feature Dependency Analysis for Product Line Component Design. In: Dannenberg, R.B., Krueger, C. (eds.) ICOIN 2004 and ICSR 2004. LNCS, vol. 3107, pp. 69–85. Springer, Heidelberg (2004)
17. Weiss, M., Esfandiari, B.: On feature interactions among Web services. In: ICWS, pp. 88–95 (2004)
18. Colman, A., Han, J.: Using role-based coordination to achieve software adaptability. Science of Computer Programming 64, 223–245 (2007)
19. Kapuruge, M.K.: Orchestration as organization. PhD Thesis. Swinburne University (2013)
20. Seidl, C., Heidenreich, F., Aßmann, U.: Co-evolution of models and feature mapping in software product lines. In: SPLC, pp. 76–85 (2012)
21. Baresi, L., Guinea, S., Pasquale, L.: Service-Oriented Dynamic Software Product Lines. Computer 45, 42–48 (2012)
22. Tarr, P., et al.: N degrees of separation: multi-dimensional separation of concerns. In: ICSE, pp. 107–119 (1999)
23. Hihn, J., Habib-agahi, H.: Cost estimation of software intensive projects: A survey of current practices. In: ICSE, pp. 276–287 (1991)
24. Bosch, J., Capilla, R.: Dynamic Variability in Software-Intensive Embedded System Families. Computer 45, 28–35 (2012)