Runtime Evolution of Multi-tenant Service Networks

Indika Kumara[1](✉), Jun Han[2], Alan Colman[2], Willem-Jan van den Heuvel[1], and Damian A. Tamburri[1]

[1] Tilburg University, Warandelaan 2, 5037 AB Tilburg, Netherlands
{I.P.K.WeerasinghaDewage,wjheuvel,d.a.tamburri}@uvt.nl
[2] Swinburne University of Technology, PO Box 218, Hawthorn, VIC 3122, Australia
{jhun,acolman}@swin.edu.au

Abstract. In a multi-tenant service network, services relate to each other and collaborate to support the functional and performance requirements of multiple tenants. Such a service network evolves over time as its services and tenants change. Consequentially, the composite application that enacts the service network also needs to evolve at runtime, which is problematic. For example, different types of changes to the application, and their consequential impacts need to be realized and managed at runtime. In this paper, we present an approach to evolving multi-tenant service networks. We identify the types of runtime changes to a service network composite application and their impacts, and present a middleware support for realizing and managing the identified changes and impacts. A software engineer can specify the desired changes to the running application, and enact the change specification to modify it. We show the feasibility of our approach with a detailed case study.

Keywords: Service network · Multi-tenancy
Change management · Evolution

1 Introduction

A business service network is a web of business services connected according to the capabilities provided and consumed by them [1,2]. The business services support business activities of enterprises or individuals (e.g., claim handling and roadside assistance). To achieve economies of scale via runtime sharing of services among tenants, a *multi-tenant service network* simultaneously hosts a set of virtual service networks (VSNs), each for a separate tenant, on the same physical service network [3,4]. The tenants generally have common and variable functional and performance requirements, and thus their VSNs share some services in the service network while also using different services as necessary.

A composite service application (hosted in a middleware runtime) can generally enact a multi-tenant business service network. It needs to connect services based on their relationships, to route and regulate the message exchanges between them, and to form VSNs over the service network. As the service network evolves, for instance, by adding a new service or tenant, or by changing the capabilities of an existing service, this composite application also needs to be evolved at runtime, which is a complex problem. Firstly, the different classes of changes that can potentially occur to the application, and the potential consequential impacts of each such change need to be identified. Secondly, a change and its impacts need to be realized and managed at runtime by the middleware without disturbing the operations of those tenants unaffected by the change.

Most existing works on service networks consider modeling and analysis of service networks from specific aspects [2] such as value flows [5], business processes [6], and service relationships [2,7]. The composite applications that enact service networks need to use service composition approaches such as BPMN (Business Process Management Notation) and BPEL (Business Process Execution Language) [1], which provide little or no direct support for the abstractions in multi-tenant service networks such as services, their relationships, message routing and regulation, and VSNs. Moreover, the existing works lack the support for two key change management activities [8] for multi-tenant service networks: identifying the impacts of a change, and realizing the change and its impacts.

In [3,4], we have proposed an approach called *Software-Defined Service Networking (SDSN)* that can deploy, enact, and manage multi-tenant service networks (composite applications). SDSN provides a programming model (a set of architectural abstractions to naturally represent a multi-tenant service network), a domain specific language (DSL), a middleware environment, and a set of tools. A software engineer can design the multi-tenant service network with the DSL, and enact and manage the designed network with the middleware at runtime.

This paper focuses on the above-mentioned two key activities of change management for a composite application that realizes a multi-tenant service network using our SDSN approach. We first identify the types of runtime changes to the application and their potential impacts. Second, we present the change management system in our SDSN middleware, including its architecture and its support for the controlled propagation of changes and impacts. The middleware also provides an ECA (event-condition-action) rules based language to specify and schedule the enactment of changes to the runtime models (*models@runtime* [9]) of the application. We present a set of guidelines that a software engineer can use to create a change specification for an evolution scenario systematically. We show the feasibility of our approach with a case study that implements common evolution scenarios for variant-rich applications (e.g., product lines and multi-tenant systems). We analyze the case study results to assess change impacts of evolution scenarios, and quantify the time taken to realize changes at runtime.

In this paper, we motivate our research and present the key requirements for a change and impact management support for multi-tenant service networks in Sect. 2. Section 3 provides an overview of our SDSN approach to realizing multi-tenant service networks. Section 4 discusses our change and impact management support in detail. Section 5 presents the prototype and evaluation of our approach. Section 6 presents related work, and Sect. 7 concludes the paper while providing the directions for further research.

2 Motivating Scenarios and General Requirements

Consider RoSAS (Road-Side Assistance Service) service network that offers roadside assistance to its tenants such as travel agencies and vehicle sellers by composing business services such as repairers and towing providers (see Fig. 1). Due to the benefits of the multi-tenancy, RoSAS shares the services among its tenants. Each tenant has a virtual service network (VSN) in RoSAS service network to coordinate roadside assistance for their users such as travelers and motorists.

The capabilities and capacities of services as well as the functional and performance requirements of tenants can exhibit commonalities and variations, which lead to the commonalities and variations in the VSNs of the tenants. For example, HappyTours and EuroCars require rental vehicle, while AnyTrucks prefers accommodation. Thus, the VSNs of the former tenants use the rental vehicle provider SilverVehicles, and the VSN of the later tenant uses the accommodation provider AmayaHotel. HappyTours' VSN uses the repairer MacRepair (for 3 days repair time) and the other two tenants' VSNs use AutoRepair (for 6 days). Compared with MacRepair, AutoRepair does not have parts internally. Thus, the VSNs of AnyTrucks and EuroCars include the part supplier JackParts. The towing provider TomTow has the limited capacity (the number of new tows per day), and cannot support the capacity requirements of both tenants AnyTrucks and EuroCars. Thus, the VSN of AnyTrucks also includes the towing provider SwiftTow. Note that the capacities of business services (e.g., towing capacity) cannot be changed by simply managing the computation resources used by them.

Let us consider two key requirements for the runtime management of the roadside assistance multi-tenant service network.

1. *Supporting Runtime Changes to Multi-tenant Service Networks.* The services and the requirements of the tenants and the service network provider can change over time. For instance, after two months, EuroCars requests the taxi

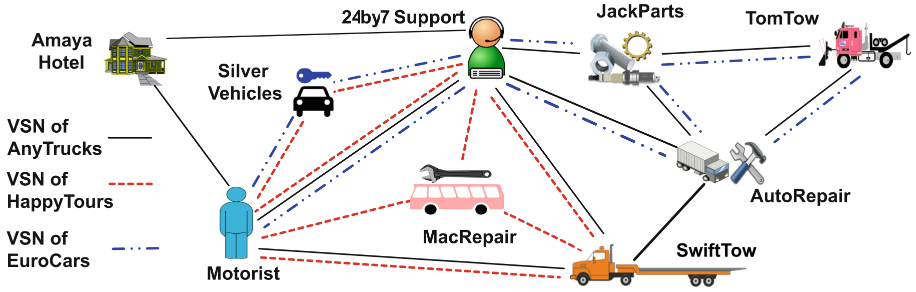


Fig. 1. Roadside assistance multi-tenant service network

hiring capability instead of the rental vehicle, and HappyTours requests 25 additional assistance cases per day. After one year, RoSAS decides to provide legal assistance for vehicle accidents. A new company starts to offer the repair assessment for vehicles, and the RoSAS provider needs to use it. To respond to or utilize these changes at the service network, a software engineer should be able to modify the RoSAS composite application at runtime. Thus, the middleware that hosts the application should support the classes of runtime changes that can occur to the application during its lifetime.

2. *Managing Consequential Change Impacts.* A change to the RoSAS composite application can further affect the application and its tenants. For example, a change to the representation of the repairer AutoRepair in the application can affect some other elements in the application and the VSNs of HappyTours and EuroCars. The middleware that hosts the composite application needs to enable a software engineer to identify such change impacts and then design and perform the controlled propagation of each change and impact.

3 Realizing Multi-tenant Service Networks: An Overview

A multi-tenant service network simultaneously hosts a set of virtual service networks on the same physical service network at runtime. In [3,4], we have proposed a novel approach, SDSN (Software-Defined Service Networking), to realize multi-tenant service networks (or cloud applications). SDSN provides a programming model, a domain specific language, and a middleware for designing and enacting multi-tenant service networks. This section provides an overview of the runtime abstractions of multi-tenant service networks in our SDSN approach.

The service network is an overlay network over the services. A *node* in the service network is a proxy to a service, and acts as a router where the messages from the other services are routed to the corresponding service via the node, and vice versa. A *link* between two nodes models the relationship between the corresponding two services, and acts as the messaging channel between the two nodes. A node has a set of *tasks* to represent the capabilities of the service. A link has a set of *interaction terms* to capture the interactions between the services.

The service network includes a set of *regulation enforcement points (REPs)* to intercept and regulate the interaction messages between services, and to monitor and enforce the performance constraints (response time and capacity) on service capabilities. There are four types of REPs: *synchronization* (at each node), *routing* (at each node), *pass-through* (at each link), and *coordinated-pass-through* (across links). The synchronization REP of a node synchronizes a subset of incoming interactions from the adjacent nodes before executing a task (sending a request to the node's service). The routing REP of a node routes a received response or request from the node's service to a subset of the adjacent nodes. The pass-through REP in a link can process the interaction messages between two nodes, and generate *events* representing the states of the interactions. The coordinated-pass-through is to regulate the interactions across different pairs of nodes. Each REP has a *knowledgebase* and a *regulation table*. The former contains event-condition-action (ECA) rules that implement regulation decisions using a number of *regulation mechanisms* such as admission control and load balance. The latter maps a message flow to a set of rules in the knowledgebase, which decide what to do with the message flow.

Each tenant has a *virtual service network (VSN)*, which is a specific service composition in the service network that meets the functional and performance requirements of the tenant. The VSNs of tenants simultaneously coexist on the same service network. Multiple business processes can exist in a VSN. Each process is a service network path, which is a subset of the service network topology. A service network path is represented by the entries in the relevant regulation tables. A table entry at a REP maps the messages belonging to a process to a subset of the regulation rules in the knowledgebase of the REP. Each such rule applies a set of regulation functions to the messages. The isolation of VSNs/processes is achieved by keeping the messages associated with a process instance isolated. Then, the isolated messages are routed and regulated on the service network path of the process instance. As the message flow continues over the network path, the business process is enacted as an event-driven business process, where events trigger the execution of tasks.

VSNs of multiple tenants share some service network elements for their common requirements, and use some other service network elements for their distinctive requirements. The elements include nodes, links, tasks, interaction terms, regulation rules/mechanisms, and services. The interested reader is referred to [3] for more details on the design and enactment of multi-tenant service networks.

Example. Figure 2 shows a part of the RoSAS service network. It consists of a number of nodes (e.g., MO, SC, and TC1) connected by links (e.g., MO-TC1, MO-SC, and SC-TC1), and supports the coordination of the interactions between the services (e.g., motorist, 24by7Support, and SwiftTow) to meet the roadside assistance requirements of the tenants. The nodes include the relevant tasks, for example, *tPickUp* of the node TC1 (to pick up a broken down vehicle). The links include the relevant interaction terms, for example, *iPickUp* of the link MO-TC1 (to represent the motorist's request for collecting the vehicle). Each

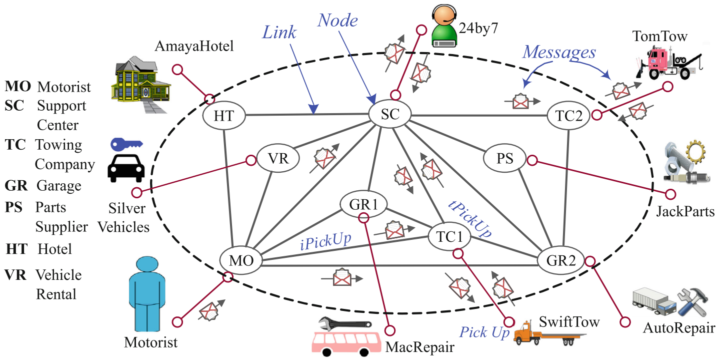
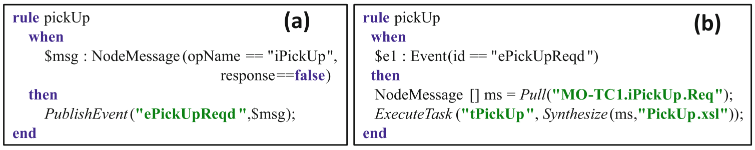


Fig. 2. Realization of the roadside assistance multi-tenant service network



(a)

```
rule pickUp
  when
    $msg : NodeMessage(opName == "iPickUp",
                       response==false)
  then
    PublishEvent("ePickUpReqd",$msg);
end
```

(b)

```
rule pickUp
 when
  $e1 : Event(id == "ePickUpReqd")
 then
  NodeMessage [] ms = Pull("MO-TC1.iPickUp.Req");
  ExecuteTask ("tPickUp", Synthesize(ms,"PickUp.xsl"));
end
```

Fig. 3. (a) a pass-through rule (link MO-TC1), (b) a synchronization rule (node TC1)

node and link also include the relevant REPs. The messages are routed and regulated over the service network via these REPs. Figure 3 shows two regulation rules. The pass-through rule generates the event *ePickUpReqd*, which triggers the synchronization rule, which creates a service request from the relevant interaction messages, and sends the request to SwiftTow to ask to collect the vehicle.

4 Change and Impact Management for Multi-tenant Service Networks

This section considers two key (runtime) change management activities [8] for multi-tenant service networks realized using our SDSN approach: (1) identifying types of changes and their consequential impacts, and (2) designing and implementing the identified changes and impacts. We discuss the types of changes and their impacts, the change management middleware system, and the process of designing a change management policy for realizing change scenarios.

4.1 Types of Changes and Impacts

A change can occur at any element of the multi-tenant service network. A given change to an element can further cause changes to that element and/or other elements as *direct consequential* impacts of the change (see Fig. 4).

Types of Changes. The *addition*, *removal*, and *update* are the three general types of changes that can occur to a given service network element. The update to an element can include the addition, removal, and update of its properties, its children elements, and its relationships with other elements. For example, an update to a node can include a change to its service endpoint reference, adding a new task, and removing a reference to a link with another node.

Types of Impacts. A direct impact of a given change to an element on another element generally depends on the type of the relationships that exist between the two elements. In a multi-tenant service network, there are four common types of relationships: (1) *containment*, (2) *association*, (3) *usage*, and (4) *representation/realization*. In the containment relationship, one element contains some other elements. For example, a node has a set of tasks. In the association relationship, one element is connected to some other elements. For example, a node is

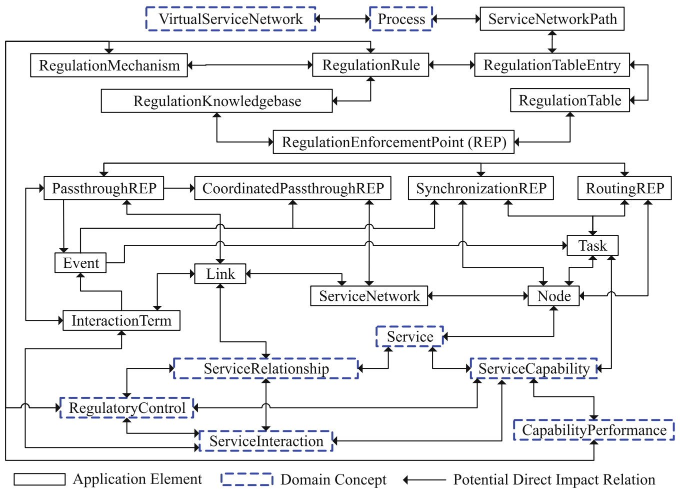


Fig. 4. Changeable elements and their potential direct impact relations

connected to another node via a link. The two nodes have references to the link. In the usage relationship, one element depends on or use some other elements for its behavior or existence. For example, an event may exist due the presence of an interaction term, and the execution of a task depends on the occurrence of some events. In the representation relationship, an element in the composite application represents a domain concept. For example, a node represents a service, and a regulation rule realizes a regulation decision.

The containment relationships include service-service capability, service relationship -service interaction/regulatory control, service capability-capability performance, service network-node/link, service network-coordinated pass-through REP, node-task, link-interaction terms, node-(routing/synchronization)REP, link-passthrough REP, REP-knowledgebase, REP-regulation table, knowledgebase-rules, regulation table-table entry, and VSN-process. The addition of a new container element generally requires new contained elements. The removal of the container element removes its contained elements. The removal of the contained elements can make the container element obsolete. For example, a new regulation knowledgebase requires new rules, the removal of a link removes its interaction terms, and the removal of the tasks of a node makes the node obsolete.

The association relationships include service-service relationship, service capability -service interaction, regulatory control-service interaction/service capability, node-link, task-interaction term, regulation rule-mechanism/event, regulation table entry-rule, and service network path-table entry. Consider the element type A and the element type B has a unidirectional association (from A to B). A new element *a1* (type A) may require an element *b1* (type B). The removal of *a1* can make *b1* obsolete if no other elements use it. The removal of *b1* makes the reference to it in *a1* dangling. For example, a new task requires the references to the interaction terms to be consumed/produced, and the removal of a referred interaction term results in a dangling reference in the task.

The usage relationships include event-interaction term, task-event, interaction term-passthrough rule, and task-routing/synchronization rule. In addition, there are mutual usage dependencies between regulation rule types. A synchronization rule requires a set of pass-through rules to analyze the interaction messages to be synchronized, and generate the events. It also requires a set of routing rules at the source nodes to initiate the interactions to be synchronized. Similarly, a routing rule has usage dependences with pass-through and synchronization rules. The events generated by a pass-through rule are generally consumed by some synchronization rules and coordinated pass-through rules. A pass-through rule also needs a routing rule to create the interaction messages that it processes.

Consider the element type *C* uses the element type *D*. A new element *c1* requires an element *d1*. The removal of the element *c1* can make the element *d1* obsolete. The removal or update of the element *d1* can adversely affect the behavior of the element *c1*. For example, the removal of a synchronization rule can make the relevant pass-through and routing rules obsolete. The removal of a pass-through rule requires the removal of or updating the conditions of the rules that use the events generated by it as those rules will not be activated.

The representation relationships include service-node, service relationship-link, service capability-task, service interaction-interaction term, regulatory control -regulation rule/mechanism, capability performance-regulation rule /mechanism (e.g., performance monitoring and admission control), and VSN/process-service network path. Consider the element type *E* realizes the domain concept *F*. The addition of the concept instance *f1* requires that of the element *e1*. The removal of *f1* makes the element *e1* invalid it represents a nonexistent concept instance. The update to *f1* may require the same to the element *e1*. For example, a new service requires a node, and the removal of an existing service makes the related node invalid as it represents a nonexistent service.

Due to the limited space, we did not provide the examples for each dependency, and each impact that the dependency creates. An interested reader may refer to an accompanying technical report [10] for more details.

4.2 Change Management System

To support the runtime changes to a multi-tenant service network in a controlled manner without compromising the consistency of the service network, we adopted the change management scheme proposed by Kramer and Magee [11]. We introduce a management state for each runtime element of a service network, which determines when an element can be removed, updated, or used. The middleware provides the capabilities to change management states, and generate the events at each state change. A software engineer can design change management policies in a way that a given change operation on an element is performed only when the element is in its appropriate management state (see Sect. 4.3).

In general, an element in a service network can be in three management states: *Active*, *Passive*, and *Quiescence*. The *Passive* state of an element enables the system to complete the existing process instances, and to move the element to its *Quiescence* state. If a runtime change to an element can adversely affect some existing process instances, then the change must be delayed until the element reaches its *Quiescence* state. A newly added element always in the *Passive* state, and must be moved to the *Active* state so that the process instances can use it. An element can be removed from the system when it is in *Quiescence* state.

VSNs, processes, and process instances can also have the above management states. A process instance for an instantiation request from a user of a tenant is created only if the VSN and its selected process is in *Active* state. Otherwise, the request message is queued, and later served when the state of the process becomes *Active*. A running instance is in the *Active* state. When it is paused, the management state becomes *Passive*, and the messages (in transit) belonging to the process instance are queued. When the management state becomes *Active*, the routing of the queued messages resumes. When the process instance is terminated, it is moved to the *Quiescence* state, and scheduled to be removed.

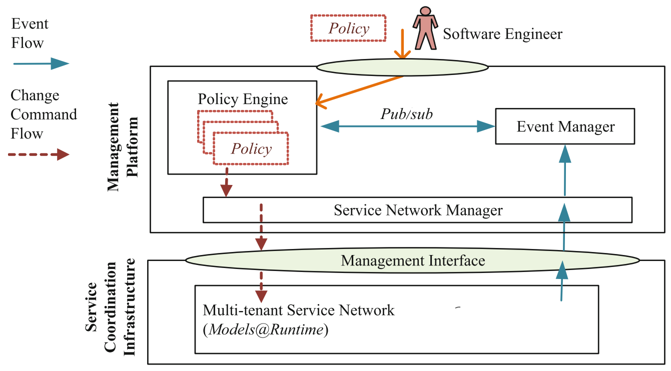


Fig. 5. Change management support in SDSN middleware

Figure 5 shows the high-level system architecture of the SDSN middleware, highlighting its change management support. It has a service coordination platform and a management platform. The former maintains multi-tenant service networks at runtime using the *models@runtime* approach, and supports the runtime changes to them (discussed in Sect. 4.1). Each change operation (e.g., *addNode* and *removeLink*) is included in the management interface, which is exposed as a Web service. The management platform includes a service network manager that uses the management Web service interface to monitor and change the running service networks, a policy engine that can maintain and enact the change management policies, and an event manager that stores the various events including management state change events. The management policies are a set of ECA rules. Generally, the conditions of a rule are events, and the actions are the change operations/commands (exposed by the management interface).

4.3 Design and Enactment of Change Management Policies

A given change can have a desired impact or an undesired impact. A direct change impact can be a consequential change or a solution (a set of intentional/designed changes) developed to utilize a desired impact, and to mitigate an undesired impact. In either case, if a change A triggers a change B as a direct impact, then to realize this impact, the change operation for propagating the change B needs to be used. A change can have a ripple effect. For example, the removal of a node may need the removal of its links with other nodes. To propagate this impact, the change operation *removeLink()* can be used. The removal of these links can have further impacts. For example, some of the tasks may refer to the interaction terms in the removed links, and the affected tasks need to be updated using the operation *updateTask()*.

A software engineer needs to identify each change and its impacts (see Sect. 4.1), and then specify them in a change management policy in terms of change operations/commands. We below provide some guidelines for a software engineer to develop such policies so that the desired changes are ordered and scheduled appropriately. We use the example of supporting the *taxi hire* feature in the RoSAS service network. A collaboration among a set of services realizes a feature.

1. *Identify and design service network topology changes.* The differences between the expected service network topology and the current one are designed in terms of (to be added or removed) nodes and links. Lines 12–14 in Fig. 6 show the topology changes for our example (a node to represent the taxi service, and two links to capture its relationships with other services).
2. *Identify and design task changes.* Next, the tasks to be added to or removed are designed (see Lines 15–16 in Fig. 6).
3. *Identify and design interaction term changes.* Next, the interaction terms to be added or removed are included (see Lines 18–19 in Fig. 6).
4. *Identify and design task-interaction changes.* The next step is to link or unlink tasks and interaction terms to reflect the required changes to the provided-required relationships in the service network. This is achieved by manipulating the inputs and outputs of the relevant tasks (see Line 21).
5. *Identify and design regulation rule and mechanism changes.* Once the modifications to the configuration design of the service network are designed, the changes to its regulation design can be introduced. These changes include regulation rule and mechanism changes at some REPs. The changes to the regulation mechanisms imply the changes to their implementations, which are Java modules in our prototype. The regulation rules can be defined as ECA rules (as .drl files) using Drools rule language (drools.org). Lines 29–31 show some of the relevant changes in our example, which add regulation rules at the node TX and the link SC-TX. Figure 7 shows two regulation rules that execute the task *orderTaxi*, and route the response from the service.
6. *Identify and design VSN changes.* The next step is to design the desired VSN changes, which include the regulation table entries at some REPs

(to be added, removed, and updated). Lines 45–47/33–35 show some relevant changes. As EuroCars replaces the rental vehicle with the taxi hire, the service network path of its VSN is modified by removing the path for the rental vehicle collaboration, and by adding the path for the taxi hire collaboration.

7. *Identify and design management state changes.* The changes to a multi-tenant service network can only be propagated when the relevant elements (to be affected) are in appropriate management states (see Sect. 4.2). The software engineer needs to initiate the appropriate state changes before and after making changes. In our example, the management state of the VSN/process is moved to the *Passive* state (see Line 5).
8. *Capture the dependencies between the individual policy rules.* As the changes need to be made to the system orderly, the software engineer needs to organize the enactment of the management policy into stages. Each stage can be represented as an ECA rule, whose conditions can use the enactment state of the policy (e.g., the end of a stage) and the management state change events. In general, we need to have stages for: (1) moving the elements to be changed to their desired management states, (2) propagating configuration design changes, (3) propagating regulation design changes, (4) removing the elements in their *Quiescence* state, and (5) moving the elements changed to their desired management states. The policy in Fig. 6 has these stages.
9. *Deploy designed management policies.* Finally, the software engineer can deploy the designed policy at the management platform. As the individual rules of the policy are executed (as their conditions are met), the changes described in the rules are propagated to the relevant runtime elements.

5 Prototype Implementation and Evaluation

Prototype Implementation. In [3,4], we have presented the prototype implementation of the SDSN approach. It includes a design language, tools, and middleware. The coordination engine and the management platform of the middleware are deployed on an Apache Tomcat web server as Apache Axis2 modules. The executable design language is XML-based, and the change management and regulation policy languages use Drools rule language. We use the Drools rule engine to implement the policy engines at the management platform and REPs. A software engineer can use the Drools IDE to define regulation rules and management policies. The SDSN implementation is available at https://github.com/road-framework/SDSN. The size of the project has 407356 lines of code (Github GLOC on 3/11/2018).

Evaluation. We show the feasibility of our approach with a case study that includes common change scenarios for multi-tenant service networks (adapted from the change scenarios for variant-rich applications [12,13]) (Table 1). To realize a scenario, we first identify the differences between the initial service

```
rule "stage1"
 when
  $mps : ManagementPolicyState (id =="policy1", state=="active")
 then
  updateProcess("EuroCars.P1","state", "passive");
  $mps.setState("stage1_done");
end
rule "stage2"
 when
  $mps : ManagementPolicyState (id =="policy1", state=="stage1_done")
 then
  addNode("TX","http://localhost:8082/axis2/services/TaxiHireService");
  addLink("SC-TX","SC","TX");
  addLink("TX-MO","TX","MO");
  addTask("TX","OrderTaxi","...");
  addTask("SC","PayTX","...");
  ...
  addTerm("SC-TX","orderTaxi","AtoB");
  addTerm("TX-MO","notifyTaxiBooking","AtoB");
  ...
  updateTask("SC","Analyze","outputs","add","SC-TX.orderTaxi.Req");
  ...
  $mps.setState("stage2_done");
end
rule "stage3"
 when
  $mps : ManagementPolicyState (id =="policy1", state=="stage2_done")
 then
 addSynchronizationRules("TX","TX_SYN.drl");
 addRoutingRules("TX","TX_Routing.drl");
 addPassthroughRules("SC-TX","SC-TX.drl");
  ...
 addSynchronizationTableEntries("EuroCars.P1","orderTaxi:TX;payTX:SC;..");
 addRoutingTableEntries("EuroCars.P1","orderTaxiRes,sendInvoice:TX;...");
 addPassthroughTableEntries("EuroCars.P1","orderTaxi,
                                        orderTaxiRes:SC-TX;...");
 $mps.setState("stage3_done");
end
rule "stage4"
 when
  $ms : ManagementState (id =="EuroCars.P1.VC.SYN", state=="quiescence") and
  $ms1 : ManagementState (id =="EuroCars.P1.VC.Routing", state=="quiescence")
  and ...
 then
 removeSynchronizationTableEntries("EuroCars.P1","VC, SC");
 removeRoutingTableEntries("EuroCars.P1","VC, SC");
 removePassthroughTableEntries("EuroCars.P1","VC-SC");
 $mps.setState("stage4_done");
end
rule "stage5"
 when
  $mps : ManagementPolicyState (id =="policy1", state=="stage3_done")  and
  $mps1 : ManagementPolicyState (id =="policy1", state=="stage4_done")
 then
  updateNode("TX","state","active");
  ...
  updateProcess("EuroCars.P1","state", "active");
  $mps.setState("quiescence");
end
```

Fig. 6. A fragment of the change management policy for our example

(a)

```
rule orderTaxi
 when
  $e1 : Event(id == "eOrderTaxiReqd")
 then
  NodeMessage [] msgs = Pull("SC-TX.orderTaxi.Req");
  ServiceMessage sMsg =
          Synthesize(msgs,"OrderTaxi.xsl");
  ExecuteTask("OrderTaxi", sMsg);
end
```

(b)

```
rule orderTaxiRes
 when
  $msg : ServiceMessage(opName== "orderTaxiRes")
 then
  Forward("SC-TX.orderTaxi.Res",
     Synthesize("OrderTaxiRes.xsl",$msg));
  Forward("TX-MO.notifyMO.Req",
     Synthesize("NotifyMO.xsl",$msg));
end
```

Fig. 7. A synchronization rule and a routing rule at the node TX

network and the target one after the realization of the scenario. Then, we design the management policy to capture the differences as change commands, and apply the created policy at the running initial service network. We compared the logs and response messages of VSN executions with those of the manually created same service network to validate the changes to the initial service network. The case study resources are at https://github.com/indikakumara/SDSN-ESOCC-2018.

We assessed the effectiveness of our support for evolution by doing a change and impact analysis. A detailed analysis of the changes and impacts for each change scenario is included in the case study resources. The scenarios together validated our support for each change type (and its impacts) to a multi-tenant service network (see Sect. 4.1). Moreover, we observed that the units of change

Table 1. Change scenarios for the roadside assistance service network

No:	Types of changes	Sub-scenarios (One functional and one performance)
1	Add/remove a mandatory feature	*Reimbursement* feature (to be used by each tenant)
		Response time <30 min and max-throughput = 150 for all assistance cases
2	Add/remove an optional feature	*Accident Tow* feature (to be used HappyTours and the new tenant AsiaBus)
		Response time <2d and throughput = 10 for a reimbursement
3	Add/remove feature to a feature group	*TaxiHire* feature to/from the features *RentalVehicle* and *PublicTransport*
		4d repair duration in addition to the existing 2d and 3d for *Repair* feature
4	Add/remove feature dependency	The dependency *Major Repair excludes Accommodation*
		The dependency *Repair time = 3d includes Tow duration = 4* h
5	Make the optional feature	*Accident Tow* a mandatory feature
		Make *Or* repair durations 2d, 3d, and 4d *Alternative (XOR)* options
6	Modify feature implementation	Extend a *Repair* implementation to use external parts if no parts available inhouse
		Add an external assessor to a repairing implementation, increasing repair time by 6 h
7	Add/remove multiple feature implementations	One realization of *Reimbursement* to/from its other realizations
		One realization of *Accident Tow* for duration = 3 h to/from its other realizations
8	Add/Remove feature implementation dependency	*Accident Tow* with MarkTow excludes *Repair* with MacRepair
		Repair duration = 3d with AutoRepair includes *RentalVehicle* duration = 3d with SLRCars

at the domain-level are generally confined to their explicit representations in the composite application, i.e., representations of services, their collaborations, their relationships, their capabilities, the routing and regulation of interactions, and VSNs/processes. This is a key requirement to support effective evolution [14].

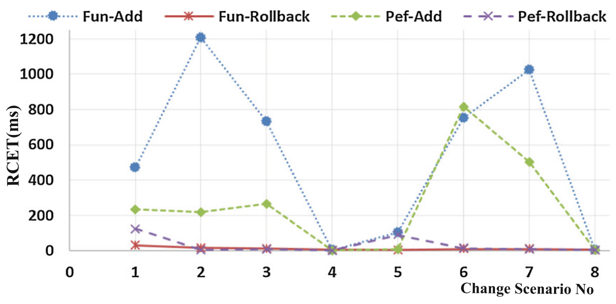


Fig. 8. Runtime change enactment time (RCET) for the change scenarios

We have also measured the run-time change enactment time (RCET) for each scenario (see Fig. 8). RCET is the time difference between the manager of the service network receiving a management policy and the service network being ready for use after applying the policy. The experiment uses a machine with an Intel i5-2400 CPU (3.10 GHz), 3.23 GB RAM, and Windows 7. The average RCET values for the functional scenarios are 537.2 ms (addition) and 12.25 ms (rollback). Those for the performance scenarios are 256.75 ms (addition) and 32.5 ms (rollback). We believe that this is reasonable.

6 Related Work

We consider the related work from variant-rich applications and service networks. We focus on the changes to the runtime artifacts of composite applications (compared with the works on design time artifacts [15] such as service specifications).

Two common types of variant rich applications are software product lines and cloud applications. Among the works from the product lines, Morin et al. [9] and Baresi et al. [12] supported modifying a business process at a set of predefined points to create variants. Bosch and Capilla [16] supported, in a smart home product line, feature-level changes by mapping a feature to a device that offers a service. The works from cloud applications considered issues such as tenant-specific variants [17,18], and tenant-specific upgrades [19]. Truyen et al. [17] used the dependency injection to bind tenant-specific variants to the variation points of a component-based cloud application. Moens et al. [18] proposed a feature-model based development of cloud applications, where a service realizes a feature. Van Landuyt et al. [19] presented a middleware support for modifying a composite cloud application by activating tenant-specific upgrades at runtime via the dynamic (re)binding of services.

In [13], we also addressed the runtime change and impact management of multi-tenant cloud applications designed as dynamic software product lines. In this paper, we considered the service network model, which is significantly different from the product-line based model in terms of runtime abstractions/elements in the composite application, and thus changes and their impacts.

Most existing works on service networks consider the modeling and analysis of service networks from specific aspects [2] such as business value flow [5], business processes [6], and service relationships [1]. Their realizations have relied on process-centric models, which fail to represent service networks naturally. That is, the domain concepts (e.g., services, service capabilities, service relationships, service interactions, interaction routing and regulation, service network paths, and virtualization) and their representations are mismatched, and the domain concepts are not directly represented or managed in the realization, limiting their utility. Regarding change and impact analysis, Kabzeva et al. [7] proposed a modeling approach to represent the entities (services, actors, and processes) in a service network, and their different relationships (e.g., consumption, competition, and ownership) at design time. They also proposed a tool to assess the impact of a change to an entity or a relationship.

Overall, there is a limited support to the runtime change and impact management for a composite service application that realizes a multi-tenant service network. The existing approaches also lack architectural abstractions to represent a multi-tenant service network naturally at runtime. They also do not provide a change and impact analysis for such runtime representations, and the middleware support for the realization and management of each change and impact. This paper addresses these limitations in the existing research.

7 Conclusions and Future Work

We have addressed the runtime evolution of a multi-tenant service network, where a single service network simultaneously hosts a set of virtual service networks for multiple tenants. We have identified different types of runtime changes to the service network and their potential impacts, and discussed our middleware support for realizing and managing those changes and impacts. A software engineer can design the controlled propagation of the desired changes. We have evaluated our approach with a case study and a performance study. The results have shown that our approach can support the runtime change and impact management of multi-tenant service networks, with little performance overhead.

In the future, we plan to develop a pattern-based formalization of the change and impact management of multi-tenant service networks, and a tool that uses the formalization to identify and assess change impacts. The generation of change management policies from high-level visual models will also be investigated.

References

1. Danylevych, O., Karastoyanova, D., Leymann, F.: Service networks modelling: an SOA & BPM standpoint. J. Univers. Comput. Sci. **16**(13), 1668–21693 (2010)
2. Razo-Zapata, P., et al.: Service network approaches. In: Handbook of Service Description: USDL and its Methods, pp. 45–274 (2012)
3. Kumara, I., et al.: Software-defined service networking: performance differentiation in shared multi-tenant cloud applications. IEEE TSC **10**(1), 9–22 (2017)
4. Kumara, I., et al.: Virtualisation and management of application service networks. In: Network as a Service for Next Generation Internet, vol. 73, pp. 357–382 (2017)
5. Allee, V.: Reconfiguring the value network. J. Bus. Strat. **21**(4), 1–6 (2000)
6. Comuzzi, M., Vonk, J., Grefen, P.: Measures and mechanisms for process monitoring in evolving business networks. Data Knowl. Eng. **71**(1), 1–28 (2012)
7. Kabzeva, A., Gtze, J., Mller, P.: Model-based relationship management for service networks. IJSSOE **5**(4), 104–132 (2015)
8. Bohner, S.A: Impact analysis in the software change process: a year 2000 perspective. In: International Conference on Software Maintenance, pp. 42–51 (1996)
9. Morin, B., et al.: Models@Runtime to support dynamic adaptation. Computer **42**(10), 44–51 (2009)
10. Kumara, I., et al.: Change and impact analysis of multi-tenant service networks. Technical report (2018). https://github.com/indikakumara/SDSN-ESOCC-2018/blob/master/TR.pdf
11. Kramer, J., Magee, J.: The evolving philosophers problem: dynamic change management. IEEE TSE **16**(11), 1293–1306 (1990)
12. Baresi, L., Guinea, S., Pasquale, L.: Service-oriented dynamic software product lines. Computer **45**(10), 42–48 (2012)
13. Kumara, I., Han, J., Colman, A., Kapuruge, M.: Runtime evolution of service-based multi-tenant SaaS applications. In: Basu, S., Pautasso, C., Zhang, L., Fu, X. (eds.) ICSOC 2013. LNCS, vol. 8274, pp. 192–206. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-45005-1_14
14. Tarr, P., et al.: N degrees of separation: multi-dimensional separation of concerns. In: International Conference on Software Engineering, pp. 107–119 (1999)
15. Andrikopoulos, V., Benbernou, S., Papazoglou, M.: On the evolution of services. IEEE TSE **38**(3), 609–628 (2012)
16. Bosch, J., Capilla, R.: Dynamic variability in software-intensive embedded system families. Computer **45**(10), 28–35 (2012)
17. Truyen, E., et al.: Context-oriented programming for customizable SaaS applications. In: ACM Symposium on Applied Computing, pp. 418–425 (2012)
18. Moens, H., Filip, T.: Feature-based application development and management of multi-tenant applications in clouds. In: SPLC, pp. 72–81 (2014)
19. Van Landuyt, D., Gey, F., Truyen, E., Joosen, W.: Middleware for dynamic upgrade activation and compensations in multi-tenant SaaS. In: Maximilien, M., Vallecillo, A., Wang, J., Oriol, M. (eds.) ICSOC 2017. LNCS, vol. 10601, pp. 340–348. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-69035-3_24