A Change Management Framework for Service Oriented Enterprises

Salman Akram[1] Athman Bouguettaya[2] Xumin Liu [3] Armin Haller[2] Florian Rosenberg[2] Xiaobing Wu[2]
Virginia Tech, USA[1] CSIRO ICT Centre, Australia[2] Rochester Institute of Technology, USA[3]
salman@vt.edu[1] {firstname.lastname}@csiro.au[2] xl@cs.rit.edu[3]

We propose a change management framework for Service-Oriented Enterprise (SOEs). We present a taxonomy of changes that occur in SOEs, whereby we focus on *bottom-up* changes. We use a combination of *Ordinary Petri nets* and *Reconfigurable Petri nets* to model the *triggering* changes and *reactive* changes, respectively. We propose an automatic change management framework that is based on the above Petri net models. We propose mapping rules and propagation algorithms that handle the *triggering* changes and *reactive* changes, respectively. We performed a simulation study to prove the feasibility of our approach.

Categories and Subject Descriptors: H.1.0 [**Models and Principles**]: General

General Terms: Life Cycle, Change Management, Reactive Model, Web Service management

Additional Key Words and Phrases: Web services, composition, change propagation, Petri net model

1. INTRODUCTION

The Web has grown from a mere repository of information to a platform for *service provision*. The Web makes it possible for businesses to share information instantly and form *virtual enterprises* to share costs, skills, and core competencies to quickly exploit worldwide sales opportunities [Hardwick and Bolton 1997]. A *virtual enterprise* [Park and Favrel 1999], describes an organizational paradigm characterized by a temporary or permanent collection of geographically dispersed entities that are dependent on electronic communication for carrying out their production process [Travica 1997]. *Virtual enterprises* provide services and products that rely on the resources of multiple businesses.

Service-oriented computing (SOC) [Papazoglou and Georgakopoulos 2003; Papazoglou et al. 2007], the prevailing software architecture style underlying modern enterprise information systems, is a key enabler of the organizational development in a Web-based *virtual enterprises*. These Web-based *virtual enterprises* are typically referred to as *Service-Oriented Enterprises* (SOEs) [Erl 2004]. In service-oriented computing functionality is provided as self-describing, platform-agnostic computational Web services that support the composition of distributed applications for enterprise application integration and collaboration [Alonso et al. 2003]. An emerging domain where SOC is applied to is Cloud computing where highly scalable services are delivered across a network [Buyya et al. 2009]. Enterprises involved in the Cloud present a set of services for consumption to end users. Unlike

the services in a SOA which are typically a set of software services, a cloud's service may be a pool of hardware, storage, data, or applications.

One of the much-touted potentials of Web services is the ability to construct SOEs on demand, relieving entrepreneurs from the intricate details of how technologies work so they can focus on the business aspects, thus unleashing a new wave of innovations in this sector.

One key aspect of SOEs is how to manage *change* to enable a full view of the integrated lifecycle of how Web services are created, modified, and disposed of. This is especially important when Web services are composed and need to evolve due, for example, to internal or external market forces.

Change management has been a popular research topic in many areas, such as software engineering, database, and workflow systems. Most of the proposed framework deal with the changes in a tightly-coupled system [Madhavji 1992; Chawathe et al. 1996; van der Aalst and Basten 2002; Shazia et al. 1999; Kradolfer and Geppert 1999; Cobena et al. 2002]. A central monitoring mechanism is usually used to detect, propagate, and react to the occurrence of changes. Our work addresses the issue of change management in SOEs. These are the results of the composition of a set of loosely-coupled Web services. No central mechanism is assumed.

As the number of services available on the Web increases, it is expected that the complexity of managing changes of SOEs will grow [Casati et al. 2003]. Any attempt to manually manage changes within a large service space would not be practical. One of the important objectives of an SOE is the ability to facilitate *long-term* and *short-term* business relationships. This require an agile and automate management of changes. For instance, SOEs must be able to dynamically plug-in or plug-out Web services with little overhead, while guaranteeing the correctness of SOE's *functional* and *non-functional* properties. The functional properties refer to the ones that are related to the functionality that an SOE fulfills. The non-functional properties refer to the events surrounding the functional properties. Change management is therefore an important research issue and a prerequisite to enable the deployment of SOEs. Our work provides such an infrastructure for change management in SOEs.

We identify two main approaches in dealing with changes:

(1) *Top-down changes:* Additionally, the business process of an SOE is usually susceptible to various changes over its lifetime that are the result of market forces and/or new legislations. In this respect, they may change their member services at will. For example, an SOE that optimizes the selection of its member services based on quality may want to replace the less quality service with the better one, thus reorganizing itself.

(2) *Bottom-up changes:* SOEs have a lifecycle that include changes, like their physical counterpart, about what and how their component services are provided. Web services can change independently their functionalities without consent of the SOEs that utilize their services. Examples are changes in a member Web service state from available to unavailable, altering the operations provided by the service, etc. Indeed, the SOE must be cognizant of any changes to its member services.

A top-down approach focuses on changes that are usually government or business mandated [Akram and Bouguettaya 2004; Liu and Bouguettaya 2007a; 2007b]. An

SOE may add a new service to its composition to take advantage of a temporal business opportunity or comply with a government regulation. The effects of these changes trickle down to the member Web services, where the changes are physically executed. Top-down changes are motivated by the SOE's business goal, and do not consider the uncertainty of the underlying member services. The treatment of this type of changes is outside the scope of this paper. The second type of changes is referred to as *bottom-up* changes because Web service providers are at the bottom of the "food chain", and are the initiators of changes. Bottom-up changes are initiated by the member services [Akram et al. 2003]. These changes are initiated in the Web service environment, and eventually translate into top-down changes of the SOE. A member service operation may become unavailable during execution and trigger the SOE to replace the service. In this paper, we focus on this aspect.

There are some existing standards providing related functionalities that can be adapted to dealing with changes of services. For example, Web Services Eventing (WS-eventing) and Web Services Notification (WSN) [OASIS 2006; W3C 2006] focus on providing an event-based framework that monitors the activities of Web services. Once an event occurs in a service, it can be sent to the other services via message passing. By affiliating a change with an event, the change on one or more member services of an SOE can be propagated to the other SOE participants. The standards can be leveraged as a tool for detecting and propagating bottom-up changes. Nevertheless, the design focus of this framework is not for change management purpose. Once a change is propagated, there is no further steps, such as identifying and reacting to the change. As a result, the existing standards are not sufficient for dealing with bottom-up changes.

We present an approach to automatic management of bottom-up changes using Petri nets. In our work, we use Petri nets to model triggering and reactive changes in SOEs. Petri nets have been used to model a variety of concurrent and discrete event distributed systems [Gracanin et al. 1993; Kristensen et al. 1998; Gou et al. 2000; Hamadi and Benatallah 2003; Iordache 2003; Llorens and Oliver 2004]. We model changes using Petri nets because of their applicability to an SOE modeling. The behavior of an SOE is described by the evolution of its Petri net model. As the Petri net evolves, the system attains different safe and unsafe states that can be completely defined by the marking of a Petri net model. For example, Petri nets readily model the states when a SOE is unsafe because of a triggering change, and the subsequent safe state achieved after managing the triggering change. Furthermore, Petri nets map directly to our change specification. They also preserve all the details of our change specification while modeling the changes accurately. For example, Petri nets can easily represent the safe an unsafe states of SOEs. They represent changes between these states as transitions. Moreover, the use of reconfigurable Petri nets allows us to incorporate our mapping rules into the Petri net model. This allows us to completely model our change specification, without the need to use additional modeling tools.

The remainder of this paper is organized as follows. In Section 2, we use a scenario from the automotive domain to motivate our work. It will also be used as a running example. Section 3 presents a bottom-up specification of changes. In Section 4, we describe our change management model which is based on Petri

nets. Section 5 presents a framework for change management. Section 6 present an extensive simulation study of the proposed framework. We provide some related work in section 7. Finally, we conclude in section 8.

2. A CAR BROKERAGE SCENARIO

As a way to motivate and illustrate this work, we use an application from the *car brokerage* domain. We categorize the SOE into two layers: *service* and *business*. The service layer consists of actual Web services, and the business layer represents the SOE business process. The business layer consists of Web service-like operations typically ordered in a particular application domain. They determine the domain of services that are required by the SOE. We refer to these services as *virtual* services. The service layer represents the Web service space. It consists of the potential services. These potential services are *a priori* unknown, and need to be discovered and matched with the virtual services at the time of SOE orchestration. We refer to the selected services as *member* services. An important feature of the virtual services is that they are not bound to any actual Web service. This is crucial since Web services are continuously evolving and the SOE is always looking for the "best" services to fulfill business requirements.

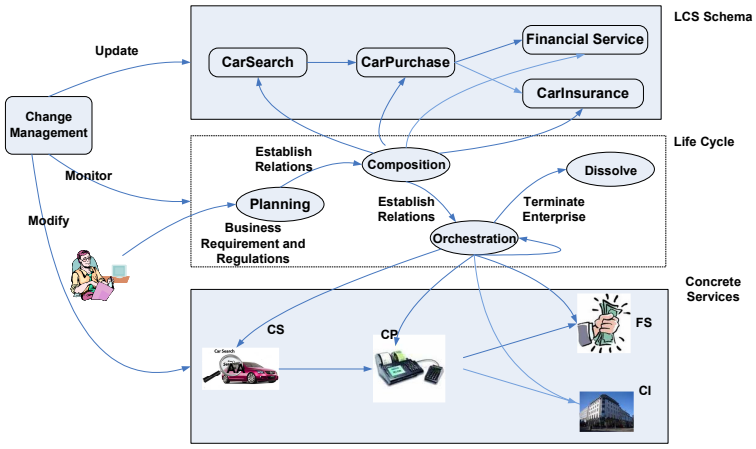

Fig. 1. A Car Brokerage Enterprise Scenario

Let us assume that an entrepreneur, say John, establishes a car agency, Enterprise Car (EC) (Figure 1). The business goal of this SOE is to provide a car brokerage package for users, including searching cars, purchasing cars, and applying for loans. During the planning phase, the following virtual services are identified: `CarSearch`, `CarPurchase`, `Financial Service`, and `CarInsurance`. Second, the entrepreneur develops a specification for EC listing the services it will compose. The third step is the orchestration of EC, where it selects and invokes the member services that match the virtual service description. We assume the `CS`, `CP`, `FS`, and CI services are selected and orchestrated. Finally, EC may disband and gracefully terminate all partnerships, or wait for another orchestration request.

This is the *ideal* sequence of events in EC's lifecycle. However, changes to member services may trigger inconsistency and uncertainty in SOE *composition* and *orchestration*. Each service layer change describes a functional or non-functional change that may occur in a member service. In turn, these changes trigger one or more reactions at the business level. An example is that the participating `carSearch` Web service may increase its price and affect the profitability of EC. In this case, EC must employ change management mechanisms and select an alternate service for the SOE to remain profitable.

3. CHANGE SPECIFICATION

Managing bottom-up changes is highly dependent on the services that compose the SOE. Therefore, it is necessary to first define the changes that occur to Web services, and then map them onto the SOE level. In this section, we define a taxonomy of bottom-up changes. We first describe the set of triggering changes followed by the set of reactive changes. We then present a mapping between these two sets of changes. Some detailed descriptions of change specifications are omitted due to space limitation. They can be found in [Akram 2005].

3.1 Taxonomy of Changes

We distinguish between service and business layer changes: *triggering* changes (δ) occur at the service level (e.g., a change to service availability) while *reactive* changes (Δ) happen at the business level (e.g., the selection of an alternative service).

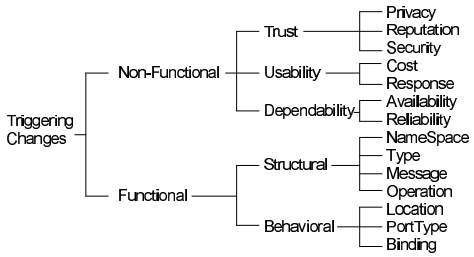

Fig. 2. Taxonomy of Triggering Changes

3.1.1 *Triggering Changes.* Changes can occur synchronously or concurrently. In this paper, we focus on dealing with changes in synchronous mode and take dealing with the concurrent change as our future work. In our scenario, we suppose that for example service `CS` may not change its data types while the triggering change of unavailability is being managed. Another assumption we make is that the service is associated with a set of states. We associate each change with a transition between two states: *precondition* and *postcondition*. In our scenario, a precondition for `CS`'s unavailability is that it was previously available and the postcondition is that it has become unavailable. Triggering changes and their respective preconditions and postconditions will be modeled using Petri nets. Our classification of triggering changes is based on the traditional approaches from the fields of software engineering and workflow systems [Madhavji 1992; van der Aalst and Basten 2002]. A

triggering change is initiated at the service level, which means that it is the result of the change of the properties of a single member service, such as the operations, the access points, the availability, etc. Therefore, we can classify triggering changes based on the properties of a Web service.

The properties of a Web service can be classified into two categories: *functional* and *non-functional*. We can thus classify triggering changes into the two categories, as depicted in Figure 2.

Non-Functional Changes. Without loss of generality, we assume that the non-functional parameters represent the *trust*, *usability*, and *dependability* aspects associated with a member service. We assume this information is maintained by an independent third party service provider. Trustworthiness of a Web service record the changes to one or more of the following aspects of a Web service: *security*, *reputation*, and *privacy*. Cost represents changes in the service *cost*. Finally, dependability is associated with changes in the *availability* and *reliability* of the Web service. Service dependability takes one of two possible values (i.e., available or unavailable). Alternatively, service trust and cost values may take more than two possible values. For instance, the service cost obviously draws its values from the real domain. Changes are triggered by values that exceed or fall below a *threshold*. This threshold consists of minimum/maximum values beyond which the SOE needs to be notified about this change. In our scenario, suppose that ET's threshold for any `airline` service cost is $10. In this case, each time a change occurs in the cost of a member `CS` service, it is compared with the threshold. Only if the change exceeds the threshold, we consider that a triggering change has occurred. Table I summarizes the non-functional changes of triggering changes.

Change	**Attribute**	δ	**Pre**	**Post**
changeAvailability	WS_A	δ_A	WS_A	WS'_A
changeReliability	WS_L	δ_L	WS_L	WS'_L
changePrivacy	WS_P	δ_P	WS_P	WS'_P
changeSecurity	WS_S	δ_S	WS_S	WS'_S
changeReputation	WS_N	δ_N	WS_N	WS'_N
changeCost	WS_C	δ_C	WS_C	WS'_C
changeResponsiveness	WS_R	δ_R	WS_R	WS'_R

Table I. Summary of Non-Functional Changes

Functional Changes. Unlike non-functional changes, which are based on attributes, functional changes deal with changes to a service's WSDL description [Christensen et al. 2001]. We represent functional changes as a combined execution of a *remove* followed by an *add*. We further classify functional changes into structural and behavioral changes (Figure 2). Structural changes refer to the operational aspects of a Web service. For example, a structural change in an airline service can be caused by changing the operations offered to a consumer. Changes to the behavior of a Web service are indicated by changing its interaction with external entities. Functional changes to a member Web service occur when its WSDL description is modified. Table II gives a summary of functional changes.

Change	**Attribute**	δ	**Pre**	**Post**
removeNameSpace	WS_N	${\delta^-}_N$	WS_N	${WS^-}_N$
addNameSpace	WS_N	${\delta^+}_N$	WS_N	${WS^+}_N$
removeType	WS_T	${\delta^-}_T$	WS_T	${WS^-}_T$
addType	WS_T	${\delta^+}_T$	WS_T	${WS^+}_T$
removeMessage	WS_M	${\delta^-}_M$	WS_M	${WS^-}_M$
addMessage	WS_M	${\delta^+}_M$	WS_M	${WS^+}_M$
removeOperation	WS_O	${\delta^-}_O$	WS_O	${WS^-}_O$
addOperation	WS_O	${\delta^+}_O$	WS_O	${WS^+}_O$
removePortType	WS_P	${\delta^-}_P$	WS_P	${WS^-}_P$
addPortType	WS_P	${\delta^+}_P$	WS_P	${WS^+}_P$
removeBinding	WS_B	${\delta^-}_B$	WS_B	${WS^-}_B$
addBinding	WS_B	${\delta^+}_B$	WS_B	${WS^+}_B$
removeLocation	WS_D	${\delta^-}_D$	WS_D	${WS^-}_D$
changeLocation	WS_D	${\delta^+}_D$	WS_D	${WS^+}_D$

Table II. Summary of Functional Changes

3.1.2 *Reactive Changes.* Reactive changes may occur at the composition and orchestration levels of an SOE. In our scenario, when ET is interrupted by a δ_A change in `CS`, it must suspend execution and react to the change. This may be accomplished by declaring a fault, compensating for the change at the composition layer, and invoking the alternate service. We split the taxonomy of reactive changes into composition and orchestration (cf. Figure 3). A third type of reactive change is referred to as userAction. We assume that each reaction may be overwritten by the user. The user always has precedence over the default changes to the business layer, and may explicitly execute a reactive change. For example, if `CS` becomes unavailable, the user may decide to replace the service, or to execute ET without `CS`. Table III gives a summary of the reactive changes defined in our model.

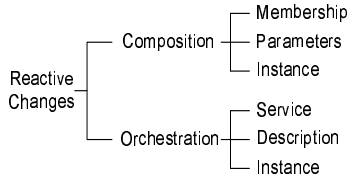

Fig. 3. Taxonomy of Reactive Changes

3.2 Mapping of Changes

A mapping specifies how change instances in one layer corresponds to changes in another layer [Velegrakis et al. 2004]. These mappings must remain consistent in the presence of frequent changes. When a change occurs at the service level, the business layer must react to manage the changes. Changes in business layer are *sympatric* and translated with respect to the service layer changes. For example, a δ change in availability maps to a Δ change of changeServiceInstance. Figure 4 maps each service level change to the corresponding business change. A dot at the

Change	**Attribute**	**Δ**	**Pre**	**Post**
removeMember	VE_M	$\Delta^-{}_M$	VE_M	$VE^-{}_M$
addMember	VE_M	$\Delta^+{}_M$	VE_M	$VE^+{}_M$
removeParameter	VE_P	$\Delta^-{}_P$	VE_P	$VE^-{}_P$
addParameter	VE_P	$\Delta^+{}_P$	VE_P	$VE^+{}_P$
removeInstance	VE_P	$\Delta^-{}_C$	VE_C	$VE^-{}_P$
addInstance	VE_C	$\Delta^+{}_C$	VE_C	VE'_C
changeState	VE_S	Δ_S	VE_S	VE'_S
changeServiceInstance	VE_I	Δ_I	VE_I	VE'_I
changeOrder	VE_O	Δ_O	VE_O	VE'_O

Table III. Summary of Reactive Changes

intersection of service and business level changes indicates a relation between the δ and Δ changes. Each δ change is also mapped directly to userAction.

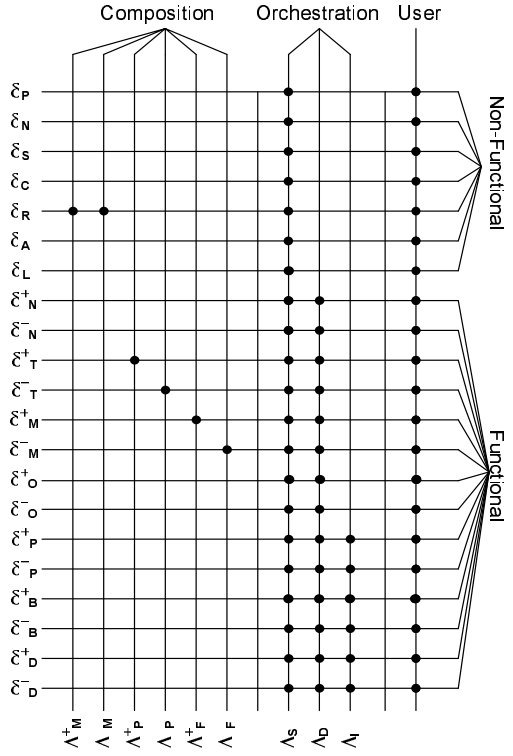

Fig. 4. Change Mapping Matrix

Our approach of mapping changes is based on mapping rules. These rules are based on the triggering changes and their corresponding reactive changes. Some changes may have more than one rule associated with them. For example, if a Web service cost increases, the SOE may continue to use the service or decide to select an alternate service because the current service has become too expensive.

Other changes may not cause any reaction. For example, if a Web service's cost decreases, the SOE will ignore this change, because it does not conflict with the service's goals. Due to space limitations we can not include the full set of mapping rules in this article and refer the interested reader to [Akram 2005].

4. CHANGE MODEL

In this section, we propose a formal change model to accurately identify the types of changes that may occur in a composite of Web services. The change model will serve as the basis of our change management framework.

4.1 Modeling Triggering Changes with Ordinary Petri Nets

Ordinary Petri nets or *OPN* are a well-founded process modeling technique that have formal semantics. They have been used to model and analyze several types of processes including protocols, manufacturing systems, and business processes. Visual representations provide a high-level, yet precise language, which allows expression and reasoning about concepts at their natural level of abstraction [Aalst 1998; Adam et al. 1998]. Services are basically a partially ordered set of changes. Therefore, it is a natural choice to map it into a Petri net. Moreover, the semantics delivered by Petri nets can be used to model the standard behavior of composite Web services described by BPEL, as well as the exceptional behavior (e.g. faults, events, compensation) [Hinz et al. 2005].

We formalize the change model for triggering changes by introducing $\mathcal{T}$-Change which is defined as follows.

Definition 4.1: *$\mathcal{T}$-Change.* $\mathcal{T}$-Change is a Petri net $\{W, \epsilon, S, i, o\}$, where:

—W is a finite set of places representing the states of a Web service
—ϵ is a finite set of transitions representing changes to Web service
—$S \subseteq (W \times \epsilon) \cup (\epsilon \times W)$ is a set of directed arcs representing a precondition and postcondition of changes in service state
—i is the input place, or starting state of the Web service
—o is the output place, or the ending state of the Web service

Figure 5 (PN_n) models non-functional changes to Web services. It consists of eight places and seven transitions. `WS` is the initial place of PN_n. It represents the initial state of the Web service (when the SOE is composed). `WS` consist of seven tokens, each representing one of the seven non-functional changes. Every time a change occurs, the corresponding token is fired. If more than one change occurs, the corresponding token for each change type is fired. For example, if a member service becomes unavailable, the transition ϵ_A will be enabled and the corresponding token will be fired.

The subnet representing dependability changes is $PN_d = (\mathrm{W}_d, \epsilon_d, \mathrm{I}_d, \mathrm{O}_d)$, where $\mathrm{W}_d = \{\mathrm{WS}, \mathrm{WS'}_L, \mathrm{WS'}_A\}$, $\epsilon_d = \{\epsilon_A, \epsilon_L\}$, $\mathrm{I}_d = \mathrm{W}_d \times \epsilon_d$, and $\mathrm{O}_d = \epsilon_d \times \mathrm{W}_d$. The place `WS` indicates that the service is both available and reliable. It contains two tokens (one for availability and the other for reliability). $\mathrm{WS'}_A$ represents a service that has become unavailable. When a service becomes unavailable, the

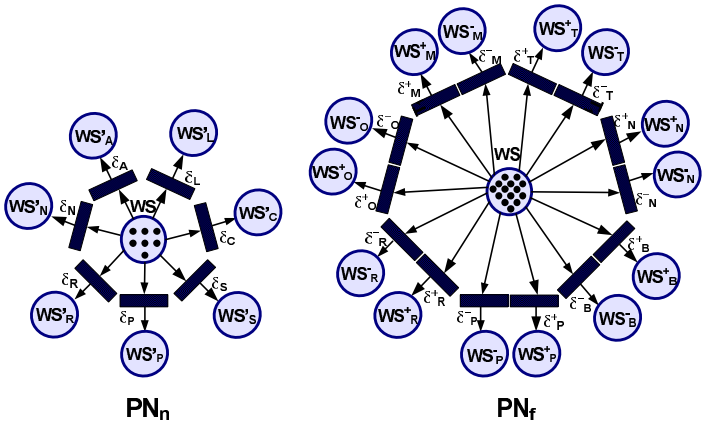

Fig. 5. Ordinary Petri Nets for Triggering Changes

availability token is moved from WS to WS'$_A$. Similarly, WS'$_L$ represents a service that has become unreliable, and the token is moved to WS'$_L$. If a service becomes both unreliable and unavailable, both the tokens are fired. The subnet PN$_d$ also consists of two transitions. ϵ_A represents the change `changeAvailability`, and ϵ_R represents `changeReliability`. Similar to changes in the dependability of a service, changes to cost and trust are also represented by a subnet of PN$_n$.

Figure 5 (PN$_f$) models functional changes to Web services. Changes to a service structure are modeled by the subnet PN$_s$ = (W$_s$, ϵ_s, I$_s$, O$_s$), where W$_s$ = {WS, WS$^-{}_N$, WS$^+{}_N$, WS$^-{}_T$, WS$^+{}_T$, WS$^-{}_M$, WS$^+{}_M$, WS$^-{}_O$, WS$^+{}_O$}, ϵ_s = {$\epsilon^-{}_N$, $\epsilon^+{}_N$, $\epsilon^-{}_T$, $\epsilon^+{}_T$, $\epsilon^-{}_M$, $\epsilon^+{}_M$, $\epsilon^-{}_O$, $\epsilon^+{}_O$}, I$_s$ = W$_s$ × ϵ_s, and O$_s$ = ϵ_s × W$_s$. Similarly, changes to service behavior are represented by the subnet PN$_b$ = (W$_b$, ϵ_b, I$_b$, O$_b$), where W$_b$ = {WS, WS$^-{}_R$, WS$^+{}_R$, WS$^-{}_P$, WS$^+{}_P$, WS$^-{}_B$, WS$^+{}_B$}, ϵ_b = {$\epsilon^-{}_R$, $\epsilon^+{}_R$, $\epsilon^-{}_P$, $\epsilon^+{}_P$, $\epsilon^-{}_B$, $\epsilon^+{}_B$}, I$_b$ = W$_b$ × ϵ_b, and O$_b$ = ϵ_b × W$_b$.

4.2 Modeling Reactive Changes with Reconfigurable PNs

An SOE can practically be modeled as a composition of Web services. This composition can be based on BPEL or another language. However, to maintain a generic and expressive model of an SOE, we use Petri nets to visualize the composition of Web services. We model each service as a place. The invocation of member services is represented by a Petri net transition. A transition also represents the return of control and data from the member service to the SOE. The execution of Web services can be modeled as a sequential, parallel, iterative, and conditional execution [Gou et al. 2000]. In our scenario, without loss of generality, we assume that the execution of Web services is sequential. That is, ET first invokes an `CS` service, receives the output, then invokes a `CP` service, and so on.

We have surveyed several extensions of Petri nets for modeling reactive changes. *Reconfigurable* PNs provide a formalism for modeling these changes. They support internal and incremental description of changes over an external and uniform description. Therefore, this type of Petri net is a natural choice for modeling reactive changes. Reconfigurable petri nets are an extension of Petri nets and a subclass of

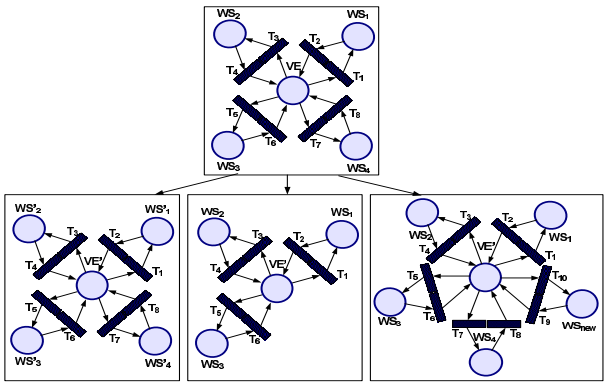

Fig. 6. RPN for Reactive Changes

net rewriting systems. They merge Petri nets with graph grammars and are best represented by Valk's Self-Modifying Nets [Llorens and Oliver 2004]:

Definition 4.10: *Reconfigurable Petri net* (*RPN*). A reconfigurable Petri net is a structure $N = (P,\ T,\ R,\ \gamma_0)$, where $P = \{p_1,...,p_n\}$ is a nonempty and finite set of places, $T = \{t_1,...,t_m\}$ is a nonempty and finite set of transitions disjoint from $P(P \cap T = \emptyset)$, $R = \{r_1,...,r_h\}$ is a finite set of rewriting rules, and γ_0 is the initial state.

We define γ_0 as the initial state when the SOE is composed, since change management will be initiated some time after the initial composition. Therefore, we say that the domain of γ_0 is SOE_0, or $DOM(\gamma_0) = SOE_0$, where SOE_0 is the initial state. In our scenario, we group the places SOE as SOE_U, SOE_V, and SOE_W. SOE_U is the set of places $\{SOE_1,\ SOE_2,\ SOE_3,\ SOE_4\}$, where U represents changeState. SOE_V is the set of places $\{SOE_5,\ SOE_6,\ SOE_7,\ SOE_8\}$, where SOE_V represents changeServiceInstance. SOE_W is the set of places $\{P_9,\ P_{10},\ P_{11},\ P_{12}\}$, where SOE_W is changeOrder. We distinguish the roles A, B, and C between transitions T. A is the set of transitions $\{T_1,\ T_2,\ T_3,\ T_4\}$, that indicate removal of a Web service, B is the set of transitions $\{T_5,\ T_6,\ T_7,\ T_8\}$ that indicate addition of a service, and C is the set of transitions $\{T_9,\ T_{10},\ T_{11},\ T_{12}\}$. Figure 6 presents an RPN representing reactive changes. It presents the initial state of the SOE (top), change in service orchestration (left), removal of service (center), and addition of service (right).

4.3 Petri Net Representation

Petri Nets are traditionally represented as an incidence matrix. $\mathcal{M}$ is an $i \times j$ matrix whose j columns correspond to the transitions and whose i rows correspond to the places of the net. $\mathcal{M}$ is generated by using the following equation:

$$\mathcal{M}_{ij} = \text{output}(\text{t}_j,\text{p}_i) \text{ - input}(\text{p}_i,\text{t}_j), \text{ for } 1 < \text{i} < \text{n}, 1 < \text{j} < \text{m}.$$

Since there are no self-loops in the triggering Petri net, we may safely create this matrix. Note that self loops in a Petri net cancel each other to yield a zero in the matrix, thus losing track of the existence of the self-loop. Since the Petri

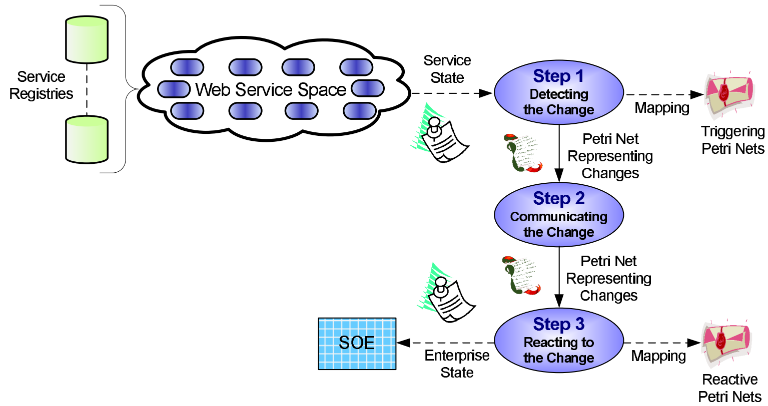

Fig. 7. Change Management Process

nets defined for changes are *sparse*, we have compressed them and use arrays as a representation. Here, we give two examples of the Petri nets that represent *removeNameSpace* and *addNameSpace* triggering changes in Table IV.

5. CHANGE MANAGEMENT

One of the most important requirements for managing changes is to guarantee consistency and correctness of an SOE in the presence of frequent changes. The SOE is likely to undergo significant functional and non-functional changes during its lifetime. Therefore, it is necessary for the SOE to automatically check for and safeguard its consistency. In this section, we present an approach to automatic management of bottom-up changes using Petri nets. The automatic management of changes is slated to play a major role in SOEs. Change management requires a systematic approach that is specifically defined and executed automatically. We use our Petri net change specification as the basis for managing changes in SOEs. We divide the process of change management into three distinct steps: *detection*, *propagation*, and *reaction* (Figure 7).

5.1 Detecting Triggering Changes

Change detection is the awareness that a change has occurred and the subsequent identification of its cause. Our approach to detecting changes is based on using agents that play the role of *monitors* and *notifiers* [Maes et al. 1999; Benatallah et al. 2000a]. These agents are Web services that monitor the participant Web services for relevant changes (e.g. functional and non-functional) and notify SOE agents about the change. Change may be detected by a periodic *pull* or *push* [Olston 2003]. In a periodic push, the Web service itself propagates changes to the service agents in

Change	Petri Net
${\delta^-}_N$	[0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
${\delta^+}_N$	[0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

Table IV. Triggering Petri Nets

regular intervals, sometimes in large batches. Alternatively periodic pulling requires that the service agents accesses remote Web services periodically to read current descriptions and update local replicas as necessary.

5.1.1 *Push Based Detection.* We assume that changes to Web services are detected through a *push-based* strategy using soft states [Raman and McCanne 1999]. *Soft states* is a method used to maintain membership of entities in a loosely coupled system, such as the SOE. This method requires that a member periodically send "refresh" messages to renew its membership. These messages are sent to a node that maintains the membership information. In our case, the provider Web services are members or participants in the SOE. The membership information is stored in the SOE *schema*. A schema is a publish/subscribe mechanism generated for each SOE orchestration instance and maintained in the SOE ontology. SOE agents act as intermediaries between the schema and the service agents. They are also responsible for the maintenance of the schema by updating it at the time of change. A participating Web service is assigned to each service agent that monitors changes in the status of that service. This agent periodically verifies the state of the service and its operations [Deolasee et al. 2002]. To verify changes in the state of a service, the agent will send "alive" messages to the Web service within a *chronon*. A chronon is the minimum granularity of time for our system [Elmasri and Navathe 2000]. Let us assume that for our example, the chronon is set to the duration between invocation of two consecutive Web services. If the Web service responds, it is assumed to be alive and its state is updated in the schema. However, if a response message is not received from the Web service within an acceptable time limit, the service is considered as unavailable. Any change to a service description (e.g., rename, change of parameters, etc.) implies that the change was made explicitly by the Web service programmers. This justifies our assumption that the Web service description in the UDDI and OWL-S registries will be appropriately updated after an operation change. We assume that these changes will further be propagated

5.1.2 *Detection and mapping rules.* We use the Petri net based model of changes to represent the detection of triggering changes. We identify the following rules for detecting functional and non-functional changes.

—For the non-functional changes, we compare the existing and new values of each attribute. If the attribute value is changed, we further compare the new value with the predefined threshold for that value.

—For the functional changes, we compare the existing and new WSDL descriptions of Web services.

Once changes have been detected by the service agent, we define the following rule for mapping the changes onto a Petri net:

Change detection mapping rule: Map the current service state as a set of precondition places in the triggering Petri net. Map the updated service state as the set of postcondition places in the triggering Petri net. Compare the values precondition and postcondition places of the Petri net. If there is a difference between any precondition and postcondition place, we place a token in the respective precondition place. This token will enable the change transition.

5.1.3 *Detection Algorithm.* The change detection algorithm (cf. Algorithm 1) takes two inputs: oldDesc and newDesc which represent the old description and new description of a Web service. Each description is composed of semantic description and syntactic description. The semantic description consists of the ontological markup for the member Web service. We assume that this information is presented in OWL-S. For detection purposes, we only utilize the extensible list of attributes defining non-functional properties of a Web service. The syntactic description of a service is presented in a WSDL format. The detection algorithm continuously checks for triggering changes. It initiates a loop for checking semantic and syntactic descriptions of each Web service. This loop is executed for each member service extracted from the composition. When a change is detected, the algorithm generates an incidence matrix for the Web service. Finally, a functional Petri net and a non-functional Petri net will be returned to represent the detected triggering changes.

Algorithm 1 ChangeDetection (Input: oldDesc, newDesc)

```
1: while newDesc do
2:    Compare (oldDesc[Functional], newDesc[Functional])
3:    if oldDesc[Functional] ≠ newDesc[Functional] then
4:       GeneratePetriNet (FunctionalPetriNet)
5:    end if
6:    Compare (oldDesc[NonFunctional], newDesc[NonFunctional])
7:    if oldDesc[NonFunctional] ≠ newDesc[NonFunctional] then
8:       Threshold = CheckThreshold (oldDesc, newDesc)
9:       if Threshold then
10:         GeneratePetriNet (NonFunctionalPetriNet)
11:      end if
12:   end if
13: end while
14: return (FunctionalPetriNet, NonFunctionalPetriNet)
```

5.2 Propagating Changes

Change propagation is required when a change in one Web service affects other entities in the SOE. In this case, all affected parties must be informed of the changes. All these changes must be managed before the SOE can arrive at a safe state. We consider two methods of change propagation in our work: *strict* and *lazy*.

Strict propagation uses a *brute-force* method of conveying changes to the SOE agent. This implies that every change that occurs in the Web service will cause the service agent to propagate the changes before the Web service is orchestrated. If the SOE finds that the service has changed, it will dump the contents of its local schema and refresh it using the latest information from the service provider. This can have some very significant performance implications, especially when the service resides on a distant network. Therefore, this method of propagation should be used only when it is absolutely necessary that the service being orchestrated is always up-to-date at the time its invocation. It also tends to completely defeat the local caching done by the SOE if there are several updates taking place concurrently on

member services. Strict change propagation does not guarantee that the member service is in its most current form. It only indicates that the next time the service is executed, the SOE will see the latest description. In our approach, we do not perform polling or background operations to constantly propagate changes. We only perform propagation when the SOE must execute the service.

Lazy change propagation is the default policy and is the most desirable in terms of efficiency and performance. Lazy change propagation works by only propagating changes when the SOE cannot locate the desired Web service. At this point, the SOE must physically re-cache the service data. Eventually, the SOE will dump its cache and retry the orchestration that it was in the process of executing when it found that the service had changed. We assume that the SOE can cache a fairly large amount of service data for that meets its requirements. This policy tends to be very efficient and will provide the best performance overall. However, it does leave the job of requesting change propagation up to the SOE agent. The amount of memory used for buffering service data can affect how often the SOE agent requests change propagation using lazy change propagation.

5.3 Reacting to Triggering Changes

In this section, we define how we execute reactive changes based on the information propagated by the service agents. First, the SOE agent receives a matrix indicating the change that has occurred. It then maps the triggering change to the appropriate reactive change. We use our mapping matrix for this conversion. Finally, the reactive change is executed by the SOE agent.

5.3.1 *Reaction Techniques.* We consider the following types of reactive changes:

—**RemoveMember:** A Web service is removed from the SOE by making its operations unavailable to the SOE. A remove partner operation followed by an add partner operation constitutes a "replace" or "change" partner operation.

—**AddMember:** A service is added in the initial composition of the SOE if a change occurs in a Web service. However, if the newly added service also has a low response rate, both the new and the old services may be retained and the requests can be distributed between them.

  —**SelectPartner:** Whenever a need for a service is determined (e.g., adding a service), the select operation identifies the required Web service. After this service is selected, it is added to the SOE.

  —**ComposeEnterprise:** The operations of the newly added Web services are plugged into the SOE.

—**changeOrchestration:** Often bottom-up changes require more than just the removal and/or addition of Web services. For example, a change in the Web service's input data type may be managed by reconfiguring the SOE input/output parameters. It is not necessary to replace the entire service based on this slight change. Therefore, the reconfigure operation updates its composition to reflect changes in member Web services.

  —**RecacheDescription:** If a service description changes, it must be reflected in the SOE composition. This will ensure successful orchestration of a member service.

—**RemovePartner:** A Web service is removed from the SOE by making its operations unavailable to the SOE. For example, if a Web service becomes permanently unavailable, it is removed from the SOE composition. A remove partner operation followed by an add partner operation constitutes a "replace" or "change" partner operation.

—**ComposeEnterprise:** The operations of the newly cached service descriptions are plugged into the SOE.

—**changeState:** When a SOE is interrupted by a stimulating change, it must change its execution phase to react to the change.

—**Fault:** If a service is not required for the SOE, the SOE may continue its orchestration. However, if a critical service becomes available, the SOE orchestration must be temporarily paused.

—**Compensate:** After a change has been detected and a fault executed, the SOE can proceed with performing a compensation function. This compensation function involves either a composition or orchestration change.

—**Invoke:** A paused SOE is resumed by the unfreeze operation. For example, if an alternate service is selected to replace an unavailable service, the SOE is immediately resumed.

—**Terminate:** If a SOA is unable to function because of missing critical services, the SOE is terminated or dissolved.

Reaction to change depends on the (i) type of change and (ii) availability of alternate services. In case of a functional change, an alternate service must be selected to fulfill the user request. The service selection stage is initiated and provided with the description of the required service. If an appropriate service does not exist (or cannot be located), the user request must be canceled. However, if an alternate service is selected successfully, it is registered with the participant list and request processing is resumed.

5.3.2 *Selecting an Alternate Service.* Reaction to change depends on the type of change and availability of alternate services. In case of a functional change, an alternate service must be selected to fulfill the user request. The service selection stage is initiated and provided with the description of the required service. At this point, the quick and dynamic discovery of alternate services is crucial to the successful execution of the SOE. If an appropriate service does not exist (or cannot be located), the user request must be canceled. However, if an alternate service is selected successfully, it is registered with the participant list and request processing is resumed.

The problem of discovering and selecting the Web services necessary to construct SOE is particularly challenging in the dynamic Web context [Doan 2002]. Discovering appropriate services dynamically is essential to fulfilling the goals of an SOE. Our approach to the problem is based on the idea of organizing Web services into ontologies. A Web service ontology is used to capture the functionality a Web service offers [Coalition 2004]. It specifies the information about the data items that a service operates on and the operations that a service offers. By the nature of ontologies, Web services can be classified into categories based on their functionalities [Medjahed et al. 2003].

Taxonomy of Web Services. The Web service space identified by an ontology may consist of more than one service that satisfies the SOE goal. From the SOE point of view, a community may consist of several candidate services.

Definition 5.1: *Candidate Service.* A candidate service is a Web service that is capable of providing the services required by a SOE but is not currently a part of the SOE. It may also be referred to as an alternate service. Formally, a Web service is a candidate service WS_c such that (i) $WS_c \subset \{WS_t\}$ and (ii) $WS_c \Rightarrow goal(VE)$ where "$\Rightarrow$" represents "satisfies" and "$goal(VE)$" is the goal of the SOE. In other words, a candidate service can be selected and integrated by a SOE in order to fulfill its goals. However, the SOE selects a single service out of these candidate services. We refer to this service as the primary service.

Definition 5.2: *Primary Service.* A primary service is the optimal service for the SOE among all the available candidate Web services and is chosen to be part of the SOE. A primary service is, thus, a Web service WS_p such that (i) $WS_c \subset \{WS_c\}$ and (ii) $\mid WS_p \mid = 1$.

5.3.3 *Reaction Algorithm.* We present the change reaction algorithm as Algorithm 2 in this section.

Algorithm 2 ChangeReaction (Input: FunctionalPetriNet, NonFunctionalPetriNet)

```
1: ReactivePetriNet = φ
2: while FunctionalPetriNet do
3:    ReactivePetriNet = Map (FunctionalPetriNet, ReactivePetriNet)
4: end while
5: while NonFunctionalPetriNet do
6:    ReactivePetriNet = Map (NonFunctionalPetriNet, ReactivePetriNet)
7: end while
8: if ReactivePetriNet then
9:    Execute (ReactivePetriNet)
10: end if
```

The *ChangeReaction* algorithm takes a functional Petri net and a non-functional Petri net as input and maps them to the reactive Petri net. The mapping function implements the semantics of the change mapping matrix as shown in Figure 4. The chosen reactive Petri net is executed in the end.

5.4 Change Management Algorithm

Now we present the change management algorithm in this section. The algorithm has two input parameters: `executionTime` and `schema`. Parameter `execuitionTime` is the time required to orchestrate an SOE. It stays positive for the SOE lifetime. Parameter `Schema` contains the list of all Web services that are currently participating in the SOE.

Within the `executionTime`, the *ChangeManagement* algorithm generates a functional Petri net and a non-functional Petri net using the *ChangeDetection* algorithm for each Web service in the `schema`. The functional and non-functional Petri nets

Algorithm 3 ChangeManagement (Input: executionTime, schema)

```
1: time = executionTime
2: while time ≠ 0 do
3:   for all Web Service WS_i in schema do
4:     (FunctionalPetriNet, NonFunctionalPetriNet) = ChangeDetection
       (oldDesc(WS_i), newDesc(WS_i))
5:     Propagate the FunctionalPetriNet and NonFunctionalPetriNet to the SOE agent
6:     ChangeReaction (FunctionalPetriNet, NonFunctionalPetriNet)
7:   end for
8:   decrement time
9: end while
10: update the schema
```

are propagated from the service agent to the SOE agent. A reactive Petri net is generated and executed using the *ChangeReaction* algorithm.

6. SIMULATION STUDY

We have chosen to perform a simulation study instead of an experimental study because it is difficult to perform change management experiments in a "real" service environment. To our best knowledge, there is not any sizeable Web service testcase available that can be used for experimental purpose. The purpose of our simulation is to assess the feasibility and correctness. The most important factor to access the feasibility and correctness of the proposed techniques in our simulation is the *accuracy* of change management. Accuracy (A_T) is defined as the level of conformity of the reaction to the triggering Petri net generated after change detection.

6.1 Simulation Setup

We utilize only services that offer airline, hotel, and car functionalities in our simulations. We assume that the Web service space consists of 100 services. Each service has two operations, therefore, a total of 200 operations are present in the services space. Furthermore, each service is utilized by one or more SOEs concurrently. The system consists of two SOEs, ET_1 and ET_2. In our work, we have defined a total of 21 triggering changes [Akram 2005]. These changes are triggered by the change simulator on behalf of the member services under a *Poisson* distribution. A Poisson process is often used to model a sequence of random events that happen independently with a fixed rate over time [Cho and Garcia-Molina 2003]. Therefore, it is ideal for simulating changes in the Web environment. Each round of simulation triggers each of the 21 changes at *least* once. The total number of changes that are triggered in each simulation are 100. Since each triggering change has a respective reactive change, the total number of reactive changes is also 100. Services that trigger a particular change are selected based on a *Zipf* distribution. The Zipf distribution is a widely used model to represent Web sites popularity [Krashakov et al. 2006]. We expect that the popularity of Web services would exhibit a similar pattern as that of Web sites. Therefore, we use the Zipf distribution to model the selection of a service from a pool of available services.

We first measure the accuracy of the change that is detected by the service agent. This detection accuracy A_D is determined by $\sum_{d=1}^{\delta(t)} \frac{A_d}{\delta(t)}$, where δ(t) is the number

of changes triggered and A_d represents the presence of the triggering change Petri net in the triggering set. Each A_d instance is either 0 or 1. It is 0 if the generated Petri net was in the triggering set and 1 if it was not present in the triggering set. Second, we measure the accuracy of the change that is generated in reaction to the triggering change. The reaction accuracy A_R is determined by $\sum_{r=1}^{\Delta(t)} \frac{A_r}{\Delta(t)}$. Each A_r instance is either a 0 or 1. It is 0 if the generated Petri net was in the reactive set and 1 if it was not present in the reactive set.

6.2 Simulation results

We did three sets of simulations to measure the accuracy of our change management methods under three different factors: different mapping relationship ratios, different number of changes and different number of member services.

The first set of simulations measure the accuracy of change management under different mapping relationships between triggering changes and reactive changes. Variable $\delta(m)$ is defined as the number of triggering changes versus the number of reactive changes that required. Four cases were simulated. The first simulation (1:1) considers the accuracy of change management when a single change is triggered and only one reactive change is required to manage this change. The second simulation (m:1) tests the accuracy of change detection and reaction when multiple triggering changes occur and require a single reactive change. Similarly, the third simulation (1:m) tests for the accuracy when one triggering change requires the execution of multiple reactive changes. Finally, the fourth simulation (m:n) tests the accuracy in the presence of multiple triggering changes that require multiple reactive changes. Figure 8 (1) depicts the results of these simulations. The results indicate that the accuracy of change detection and reaction is 100% if there is one-to-one or many-to-one mapping between the triggering and reactive changes. However, when there is a one-to-many or many-to-many relationship between the triggering and reactive changes, the accuracy drops to 82%. The reason for reduced accuracy is that the change management framework always executes the *first* reactive change indicated by the reactive net. Since 82% of the one-to-many and many-to-many changes require a selection of alternate service, their reaction is executed accurately. For others, the reaction is invalid and does not comply with our change mapping rules. For example, if a triggering change of availability occurs, the reaction is to either select an alternate service, or to execute the SOE without the service. Since selecting an alternate service is hierarchically above the service deletion reaction, the reaction to service unavailability is always to select an alternate service. In our future work, we plan to prioritize changes and determine a reaction that will result in the optimal SOE.

We assume that all changes triggered in these set of simulation have a one-to-one mapping of changes.

Figure 8 (2) depicts the results of these simulations. The results of these simulations indicate a slight drop in accuracy as the number of changes increase. For example, when the number of changes are 100, the accuracy of change management is 100%. As the number of changes approaches to 500, the accuracy drops to 98%. At 2000 changes, the accuracy of our approach is measured at 81%. This drop in accuracy is attributed to the highly volatile nature of ET during change man-

agement. First, ET and its member services' attributes are constantly changing. Because changes occur simultaneously, it is possible that the state of the service retrieved by the service agent is not accurate. Second, ET's membership will change over time. For example, if `CS` becomes unavailable and no alternate service exists, ET will orchestrate without `CS`. In this case, any change generated by the change simulator that references to an airline service is inherently inaccurate.

The third set of simulations measure the accuracy of our change management approach as the number of member service increases from 1 to 12. We consider the number of changes to remain constant at 100. Also, the mapping of changes is set as one-to-one. Figure 8 (3) depicts the results of these simulations. The simulations indicate that the accuracy of change management remains optimal when the number of member services is between 1 and 3. However, the accuracy drops to slightly below optimal when N_{ms} is 6. The accuracy further decreases when N_{ms} is 12. The increase of the number of member services implies a high probability of changes. This introduces the drop of the accuracy.

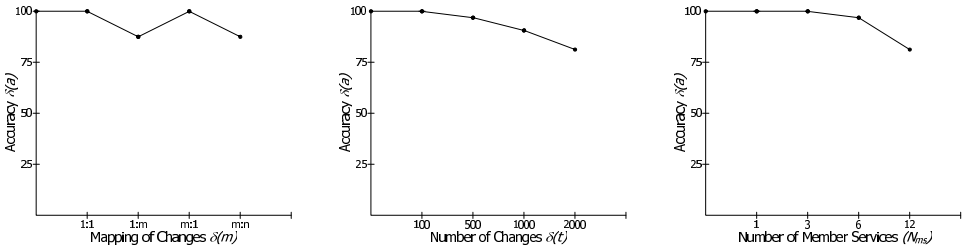

Fig. 8. Accuracy under different mapping ratios, different number of changes and different number of member services.

7. RELATED WORK

Researchers have long been interested in enabling adaptive, market-driven enterprises [Benatallah et al. 2000b]. The focus has over the years shifted from the traditional single-entity enterprises to service-based enterprises [Petrie and Bussler 2003; Benatallah et al. 2000a; Casati et al. 2000]. Various standards efforts have already been adopted to facilitate the integration of Web services at different levels through *compositions*, *orchestrations*, and *choreographies* [Weerawarana et al. 2005]. However, managing the changes in the lifecycle of composed Web services has received little attention in the standardization community. Exceptions are WS-Eventing [OASIS 2006] and WS-Notification [W3C 2006] that provide some syntactic support for change notification. Composition standards mostly deal with changes as exceptions, much like workflow technology does. For example, WS-BPEL [OASIS 2006], a composition language for Web services, handles changes as exceptions, usually by invoking the `fault` method. This traditional, workflow-based approach does not offer an adequate change management solution for SOEs. Therefore, most changes in BPEL-based SOEs are manually managed. The challenge is to automate the change management process to make it transparent to users.

Workflows are the popular means of composing enterprises. They provide the ability to execute business processes that span multiple organizations [Tagg 2001]. Traditional workflows do not provide methods for dynamic change management. Workflows are geared towards *static* integration of components. This characteristic inhibits the profitability, adaptability, utility, and creativity in an enterprise. Furthermore, workflows do not cater for the behavioral aspects of Web services. For example, they do not distinguish between the internal and external processes of a Web service [Bussler 2003].

In [Rinderle et al. 2004], a Petri net based approach to maintaining correctness between process *type* and *instances* is presented. A process type represents a particular business process described by a schema. Process instance is a real time execution of the process type. Changes to process type occur when the process schema is modified in response to the environment. For example, a business process may adapt to comply with new legislation, or it may be optimized for performance reasons. The respective process type changes must be propagated to the process instances. Inversely, process instances may be changed to accommodate for changes in the execution environment. For example, an exception may cause the process to skip a task. A mapping of this instance to the process type results in a schema that is different from the original process schema. When process type and instance changes are executed independently, they are no longer in harmony with each other.

In [Brambilla et al. 2005], a framework to detect and react to the exceptional changes that can be raised inside workflow-driven Web application is proposed. It first classifies these changes into *behavioral (or user-generated)*, *semantic (or application)*, and *system* exceptions. The behavior exceptions are driven by improper execution order of process activities. For example, the free user navigation through Web pages may result in the wrong invocation of the expired link, or double-click the link when only one click is respected. The semantic exceptions are driven by unsuccessful logical outcome of activities execution. For example, a user does not keep paying his periodic installments. The system exceptions are driven by the malfunctioning of the workflow-based Web application, such as network failures and system breakdowns. It then proposes a modeling framework that describes the structure of activities inside hypertexts of a Web application. The hypertext belonging to an activity is broken down into pages, where are univocally identified within an activity. It proposes a framework to handle these changes. The framework consists of three major components: *capturing model*, *notifying model*, and *handling model*. The capturing model capture events and store the exceptions data in the workflow model. The notifying model propagate the occurred exceptions to the users. The handling model defines a set of recovery policy to resolve the exception. For different types of exceptions, different recovery policies will be used.

In [Ryu et al. 2008], the work proposes a framework that manages the business protocol evolution in service-oriented architecture. It uses several features to handling the running instances under the old protocol. These features include *impact analysis* and *data mining based migration analysis*. The impact analysis is to analyze how protocol change impacts on the running instances. It will be used to determine whether ongoing conversations are *migrateable* to the new protocol or

not. The data mining based migration analysis is used for cases where the regular impact analysis cannot be performed. Service interaction logs are analyzed using data mining techniques. It then uses the result of the analysis to determine whether a conversion is *migrateable* or not.

In [Ellis and Keddara 2000], the work focuses on modeling dynamic changes within workflow systems. It introduces a Modeling Language to support Dynamic Evolution within Workflow System (ML-DEWS). A change is modeled as a process class, which contains the information of *roll-out time*, *expiration time*, *change filter*, and *migration process*. The roll-out time indicates when the change begins. The expiration time indicates when the change ends. The change filter specifies the old cases that are allowed to migrate to the new procedure. The migration process specifies how the filtered-in old cases migrate to the new process.

A large body of work is proposed within the scope of autonomic computing, in particular self-adaptation research [Salehie and Tahvildari 2009]. The focus is on models and mechanisms that allow a software system to modify its own behavior in response to changes in the operating environment [Oreizy et al. 1999]. Recently, this principle has also been applied to SOEs to transparently handle top-down and bottom up changes.

In [Moser et al. 2008], a non-intrusive WS-BPEL extension called VieDAME is proposed. It can adapt services in a WS-BPEL process based on different selection strategies. For example, it can adapt the process to replace an existing serivce with a service that has the best response time or the lowest failure rate. The adapation is based on the assumption that a transformation rule from the service currently bound in the WS-BPEL process to the alternative service exists. In contrast to our work, VieDAME only provide a limited number of bottom-up changes, in particular non-functional changes. It does not facilitae explicit support for change management because the runtime transparently handles the changes.

In [Michlmayr et al. 2010], a QoS-aware middleware VRESCo is presented that addresses adaptability of service-oriented applications. They propose several infrastructure components and services including dynamic binding and invocation, querying, eventing and composition that are based on a unified metadata model. Applications built on top of VRESCo are then able to handle several changes types automatically. These include non-functional changes leading to a service replacement in a composition or service versioning in case a new version of a service is available. In contrast to our work, VRESCo does not provide explicit and automated support for various changes types. The focus of VRESCo is on non-functional changes whereas this paper focuses on both change types within a unified framework.

8. CONCLUSION

We have identified a taxonomy of changes in SOEs using a bottom-up approach. In this approach, we first describe triggering changes that may occur in Web services. These changes are then mapped to reactive changes in SOEs. We propose a formal change model based on Petri nets to accurately represent these changes. We describe a change management framework based on our change model to provide automatic management of changes in SOEs. Finally, we conduct an extensive

simulation study to prove the feasibility of the proposed techniques.

Future work includes extending our change management approach. We plan to include a top-down approach to specifying changes. Top-down changes are motivated by the dynamic business environment. These changes are usually voluntary, and in reaction to changes in the business environment. A prime application of top-down change management is to modify SOE membership to increase competitiveness, efficiency, or customer base. Using a top-down approach, we will first analyze the types of changes that are initiated at the business level. We will then translate and map those changes to the Web service level.

Change management in SOEs presents vast opportunities for research. This paper attempts to initiate work in this field. Future work includes investigating the following issues: prioritizing changes, changes to service semantics, top-down changes, cascading changes, verifying changes, estimating frequency of changes, and preserving mapping consistency between triggering and reactive changes.

REFERENCES

Aalst, W. 1998. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers 8,* 1, 21–66.

Adam, N. R., Atluri, V., and Huang, W.-K. 1998. Modeling and analysis of workflows using petri nets. *J. Intell. Inf. Syst. 10,* 2, 131–158.

Akram, M. S. 2005. Managing Changes to Service Oriented Enterprises. M.S. thesis, Virginia Polytechnic Institute and State University, Falls Church, Virginia, USA. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.106.7479&rep=rep1&type=pdf`.

Akram, M. S. and Bouguettaya, A. 2004. Managing Changes to Virtual Enterprises on the Semantic Web. In *Fifth International Conference on Web Information Systems Engineering.* Brisbane, Australia, 472–478.

Akram, M. S., Medjahed, B., and Bouguettaya, A. 2003. Supporting Dynamic Changes in Web Service Environments. In *First International Conference on Service Oriented Computing.* Trento, Italy, 319–334.

Alonso, G., Casati, F., Kuno, H., and Machiraju, V. 2003. *Web services: Concepts, architectures and applications.*

Benatallah, B., Medjahed, B., Bouguettaya, A., Elmagarmid, A., and Beard, J. 2000a. Composing and maintaining web-based virtual enterprises. In *First VLDB Workshop on Technologies for E-Services.* Cairo, Egypt.

Benatallah, B., Medjahed, B., Bouguettaya, A., Elmagarmid, A. K., and Beard, J. 2000b. Composing and maintaining web-based virtual enterprises. In *TES.* 155–174.

Brambilla, M., Ceri, S., Comai, S., and Tziviskou, C. 2005. Exception handling in workflow-driven web applications. In *WWW '05: Proceedings of the 14th international conference on World Wide Web.* ACM Press, New York, NY, USA, 170–179.

Bussler, C. 2003. The Role of Semantic Web Technology in Enterprise Application Integration. *Data Engineering Bulletin 26,* 4 (December), 62–68.

Buyya, R., Yeo, C. S., Venugopal, S., Broberg, J., and Brandic, I. 2009. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems 25,* 6, 599 – 616.

Casati, F., Ilnicki, S., Jin, L., Krishnamoorthy, V., and Shan, M.-C. 2000. Adaptive and Dynamic Service Composition in eFlow. In *CAiSE Conf.* Stockholm, Sweden, 13–31.

Casati, F., Shan, E., Dayal, U., and Shan, M. 2003. Business-oriented management of web services. *Communications of the ACM 46,* 10 (October), 55–60.

Chawathe, S. S., Rajaraman, A., Garcia-Molina, H., and Widom, J. 1996. Change detection in hierarchically structured information. In *SIGMOD.* Montreal, Canada, 493–504.

Cho, J. and Garcia-Molina, H. 2003. Estimating frequency of change. *ACM Transactions on Internet Technology 3,* 3 (August), 256–290.

Christensen, E., Curbera, F., Meredith, G., and Weerawarana, S. 2001. Web Services Description Language (WSDL) 1.1. Tech. rep., W3C, https://www.w3.org/TR/wsdl. March.

Coalition, T. O. S. 2004. Owl-s: Semantic markup for web services. Tech. rep., http://www.daml.org/services/owl-s/1.1B/owl-s/owl-s.html. July.

Cobena, G., Abiteboul, S., and Marian, A. 2002. Detecting changes in xml documents. In *Proceedings of the 18th International Conference on Data Engineering*. San Diego, USA, 41–52.

Deolasee, P., Katkar, A., Panchbudhe, A., Ramamritham, K., and Shenoy, P. 2002. Adaptive Push-Pull: Disseminating Dynamic Web Data. *IEEE Transactions on Computers 51,* 6.

Doan, A. 2002. Learning to Map between Structured Representations of Data. Ph.D. thesis, University of Washington.

Ellis, C. A. and Keddara, K. 2000. A workflow change is a workflow. In *Business Process Management, Models, Techniques, and Empirical Studies*. Springer-Verlag, London, UK, 201–217.

Elmasri, R. and Navathe, S. B. 2000. *Fundamentals of Database Systems - Third Edition*. Addison-Wesley, Reading, Massachusetts.

Erl, T. 2004. *Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services*. Prentice Hall, Upper Saddle River, NJ.

Gou, H., Huang, B., Liu, W., Ren, S., and Li, Y. 2000. Petri net based business process modeling for virtual enterprises. In *IEEE International Conference on Systems, Man, and Cybernetics*. Nashville, United States, 3183–3188.

Gracanin, D., Srinivasan, P., and Valavamis, K. 1993. Fundamentals of parameterized petri nets. In *International Conference on Robotics and Automation*. Atlanta, USA, 584–591.

Hamadi, R. and Benatallah, B. 2003. A petri net-based model for web service composition. In *Proceedings of the Fourteenth Australasian database conference on Database technologies*. Australian Computer Society, Inc., 191–200.

Hardwick, M. and Bolton, R. 1997. The Industrial Virtual Enterprise. *Commun. ACM 40,* 9, 59–60.

Hinz, S., Schmidt, K., and Stahl, C. 2005. Transforming BPEL to Petri Nets. In *Proceedings of the Third International Conference on Business Process Management (BPM 2005)*, W. M. P. v. d. Aalst, B. Benatallah, F. Casati, and F. Curbera, Eds. Lecture Notes in Computer Science, vol. 3649. Springer-Verlag, Nancy, France, 220–235.

Iordache, M. V. 2003. Methods for the supervisory control of concurrent systems based on petri net abstractions. Ph.D. thesis, University of Notre Dame.

Kradolfer, M. and Geppert, A. 1999. Dynamic workflow schema evolution based on workflow type versioning and workflow migration. In *Conference on Cooperative Information Systems*. 104–114.

Krashakov, S. A., Teslyuk, A. B., and Shchur, L. N. 2006. On the universality of rank distributions of website popularity. *Comput. Netw. 50,* 11, 1769–1780.

Kristensen, L. M., Christensen, S., and Jensen, K. 1998. The practitioner's guide to coloured petri nets. *International Journal on Software Tools for Technology Transfer 2,* 1, 98–132.

Liu, X. and Bouguettaya, A. 2007a. Managing top-down changes in service-oriented enterprises. In *IEEE International Conference on Web Services 2007*. UTAH, USA.

Liu, X. and Bouguettaya, A. 2007b. Reacting to functional changes in service-oriented enterprises. In *CollaborateCom 2007*. White Plains, NY.

Llorens, M. and Oliver, J. 2004. Structural and dynamic changes in concurrent systems: Reconfigurable petri nets. *IEEE Transactions on Computers 53,* 9 (September), 1147–1158.

Madhavji, N. H. 1992. Environment evolution: The prism model of changes. *IEEE Trans. Softw. Eng. 18,* 5, 380–392.

Maes, P., Guttman, R. H., and Moukas, A. G. 1999. Agents that Buy and Sell. *Communications of the ACM 42,* 3 (March), 81–91.

Medjahed, B., Bouguettaya, A., and Elmagarmid., A. 2003. Composing Web Services on the Semantic Web. *The VLDB Journal, Special Issue on the Semantic Web 12,* 4 (November).

Michlmayr, A., Rosenberg, F., Leitner, P., and Dustdar, S. 2010. End-to-End Support for QoS-Aware Service Selection, Binding and Mediation in VRESCo. *IEEE Transactions on Services Computing*. (to appear).

Moser, O., Rosenberg, F., and Dustdar, S. 2008. Non-Intrusive Monitoring and Service Adaptation for WS-BPEL. In *Proceeding of the 17th International Conference on World Wide Web (WWW'08), Beijing, China*. ACM, 815–824.

OASIS 2006. *Web Service Business Process Execution Language 2.0.* OASIS. URL: `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel` (Last accessed: Apr. 17, 2007).

OASIS. 2006. Web Services Eventing (WS-Eventing). `http://www.w3.org/Submission/WS-Eventing/`.

Olston, C. A. R. 2003. Approximate Replication. Ph.D. thesis, Stanford University.

Oreizy, P., Gorlick, M. M., Taylor, R. N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D. S., and Wolf, A. L. 1999. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems 14*, 54–62.

Papazoglou, M. P. and Georgakopoulos, D. 2003. Service-Oriented Computing. *Commun. ACM 46,* 10, 25–28.

Papazoglou, M. P., Traverso, P., Dustdar, S., and Leymann, F. 2007. Service-Oriented Computing: State of the Art and Research Challenges. *IEEE Computer 40,* 11, 38–45.

Park, K. H. and Favrel, J. 1999. Virtual enterprise – Information system and networking solution. *Computers & Industrial Engineering 37,* 1-2, 441–444.

Petrie, C. and Bussler, C. 2003. Service Agents and Virtual Enterprises: A Survey. *IEEE Internet Computing 7,* 4 (July-August), 68–78.

Raman, S. and McCanne, S. 1999. A model, analysis, and protocol framework for soft state-based communication. *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*.

Rinderle, S., Reichert, M., and Dadam, P. 2004. On Dealing with Structural Conflicts Between Process Type and Instance Changes. In *Second International Conference on Business Process Management*. Postdam, Germany, 274–289.

Ryu, S. H., Casati, F., Skogsrud, H., Benatallah, B., and Saint-Paul, R. 2008. Supporting the dynamic evolution of Web service protocols in service-oriented architectures. *ACM Trans. Web 2,* 2, 1–46.

Salehie, M. and Tahvildari, L. 2009. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems 4,* 2, 1–42.

Shazia, S., Olivera, S., Maria, M., and Orlowska, E. 1999. Managing change and time in dynamic workflow processes. *International Journal of Cooperative Information Systems*.

Tagg, R. 2001. Workflow in Different Styles of Virtual Enterprise. In *Workshop on Information technology for Virtual Enterprises*. Queensland, Australia, 21–28.

Travica, B. 1997. The design of the virtual organization: a research model. In *Proceedings of the Americas Conference on Information Systems (AMCIS 1997)*. Indianapolis, IN, USA, 134–143.

van der Aalst, W. M. P. and Basten, T. 2002. Inheritance of workflows: an approach to tackling problems related to change. *Theor. Comput. Sci. 270,* 1-2, 125–203.

Velegrakis, Y., Miller, R. J., and Popa, L. 2004. Preserving Mapping Consistency Under Schema Changes. *The VLDB Journal 13,* 3 (September), 274–293.

W3C. 2006. Web Services Notification (WSN). `http://www.oasis-open.org/committees/tc\_home.php?wg\_abbrev=wsn`.

Weerawarana, S., Curbera, F., Leymann, F., Storey, T., and Ferguson, D. F. 2005. *Web Services Platform Architecture : SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall PTR.