Facing Uncertainty in Web Service Compositions

Germán H. Alférez
Facultad de Ingeniería y Tecnología,
Universidad de Montemorelos,
Apartado 16-5, Montemorelos N.L., 67500, Mexico
harveyalferez@um.edu.mx

Vicente Pelechano
Centro de Investigación en Métodos
de Producción de Software (ProS),
Universitat Politècnica de València,
Camí de Vera s/n, E-46022, Spain
pele@dsic.upv.es

***Abstract*—Web service compositions run in complex computing infrastructures where arising events may affect the quality of the system. However, crucial Web service compositions cannot be stopped to apply changes to deal with problematic events. Therefore, the trend is moving towards context-aware Web service compositions, which use context information as a basis for autonomic changes. Under the closed-world assumption, the context and possible adaptations are fully known at design time. Nevertheless, it is difficult to foresee all the possible situations arising in uncertain contexts. In this paper, we leverage models at runtime to guide the dynamic evolution of context-aware Web service compositions to deal with unexpected events in the open world. In order to manage uncertainty, a model that abstracts the Web service composition, self-evolves to preserve requirements. The evolved model guides changes in the underlying WS-BPEL composition schema. A prototype and an evaluation demonstrate the feasibility of our approach.**

***Keywords*-Uncertainty, Web service compositions, models at runtime, dynamic evolution, open world.**

I. Introduction

Today's systems run in complex and heterogeneous computing infrastructures in which a diversity of events may arise (e.g. security threats and server failures). Therefore, it is desirable to count on self-adjusting mechanisms to solve these situations. A good example of systems that require adjusting themselves are the ones based on Web service compositions (hereafter, service compositions). In an ideal scenario, Web service operations would do their job smoothly. However, several exceptional situations may arise in the changing contexts where they run. For example, the response time of a Web service operation may have greatly increased. Therefore, it is appropriate to count on context-aware service compositions that dynamically change to keep service-level agreements (SLAs), offer extra functionality, protect the system, or make the system more usable. *Dynamic adaptation* refers to the act of changing the software behavior as it executes, without stopping it. This type of adaptation is specially important in critical service compositions that cannot be stopped to implement adaptations.

Although current research works have paved the way towards the dynamic adaptation of service compositions, they have two main drawbacks: 1) most solutions have tended to implement dynamic adaptations with variability constructs at the language level [1], [2], [3], [4] (e.g. by extending WS-BPEL code). However, this approach makes it harder to understand and communicate autonomic behavior decisions among stakeholders. This may result in error-prone systems. Moreover, any change in the platform or the application may require to change low-level platform-specific scripts that are tightly coupled with the application code. This process may become tedious and time-consuming; 2) dynamic adaptations are carried out in the *closed world*, in which the boundary between system and context is known ahead and unchanging [5]. In this scenario, a set of adaptation actions is predefined for fully foreseen context events [6], [7], [8]. However, in the unpredictable *open world*, service compositions should react to continuous and unanticipated changes in the context.

In the open world, *uncertainty* is caused by how the service composition should deal with unknown context events. *Unknown context events* are those situations in the context that have not been foreseen at design time [9].

In this paper, we try to manage some situations of uncertainty in the open world by self-evolving service compositions through models at runtime. *Models at runtime* are causally connected self-representations of the associated system that emphasize the structure, behavior, or goals of the system from a problem space perspective [10]. At the time unknown context events arise during execution, our approach triggers the *dynamic evolution* of the service composition to manage these events properly. To this end, easy-to-understand and technology-independent models are used to describe self-adjusting actions that preserve the expected requirements. The evolved models guide changes in the underlying WS-BPEL composition schema, which orchestrates the service composition. As a result, dynamic evolutions move the service composition to new versions, which cannot be supported by predefined dynamic adaptations. Since we are interested in managing uncertainty that arises from the context in which the service composition is deployed, our approach is related to external uncertainty [11].

The remainder of this paper is structured as follows: Section 2 describes a running example. Section 3 summarizes the dimensions of dynamic adaptation and dynamic evolution in service compositions. Section 4 presents an overview of our approach for dynamic evolution of service compositions. Section 5 describes the models that guide evolutions. Section 6 summarizes the mechanisms that support evolution. Section 7 describes a prototype that realizes our approach and evaluation results. Section 8 presents related work. Section 9 presents

conclusions and future work.

II. Running Example

In order to illustrate the need for dealing with uncertainty in the open world, we introduce a critical service composition for online book shopping. The example is specified with the Business Process Model and Notation (BPMN) in Figure 1. BPMN tasks express Web service operations (e.g. UPS Shipping); and BPMN subprocesses express composite service operations (e.g. Barnes & Noble Books).

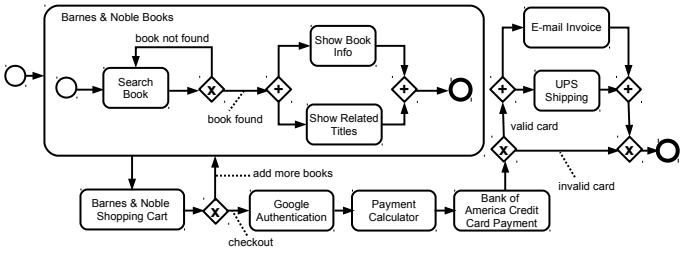

Figure 1. A BPMN model that represents a composite service for online book shopping

The business process (BP) starts when a customer looks for a book on the store's website. The first thing the customer wants to do is to identify the books to purchase. The searching operation is provided by the Search Book Web service, which is part of the Barnes & Noble Books composite service. When a book is found, then the book information is returned to the customer by the Show Book Info Web service while at the same time the information for other related books is listed by the Show Related Titles Web service. If no book is found, then the customer must refine the search, e.g. using supplementary or different search criteria, or undertaking another search. In the next step, the customer adds books into the shopping cart through the Barnes & Noble Shopping Cart Web service. The process can start over again until the customer is satisfied with his or her selection. When the customer is ready to checkout, he or she has to be authenticated by the Google Authentication Web service. Then, the in-house Payment Calculator Web service calculates the total amount to be paid. The payment is done through the Bank of America Credit Card Payment Web service. Finally, if the credit card information is valid, the in-house E-mail Invoice Web service sends an e-mail to the customer with the invoice while the UPS Shipping Web service is invoked to deliver the book. Otherwise, the process terminates.

A set of adaptation actions have been created for foreseen context events. For instance, if the Barnes & Noble Books composite service operation is unavailable, then other service operations can be invoked instead. Nevertheless, if there are no predefined adaptation actions for unknown context situations (e.g. any third-party Web service operation fails or performs below required SLAs), then no adaptation is carried out. As a result, the whole system quality may be harmed. We argue that despite unknown context events, the service composition has to keep offering expected requirements.

III. Dimensions of Dynamic Adaptation and Dynamic Evolution in Service Compositions

In order to introduce our approach, first it is necessary to describe the dimensions of dynamic adaptation and dynamic evolution in service compositions. Figure 2 summarizes these dimensions. The two dimensions in the lower section are about *what should be changed* according to information retrieved from the context. In the dimension of *dynamic adaptation of running instances*, only the instance that has triggered the adaptation is adapted. In contrast, the dimension of *dynamic evolution of the composition schema* modifies the composition schema (e.g. described in WS-BPEL), and therefore all future instances. These two dimensions have been widely described in literature [12], [13].

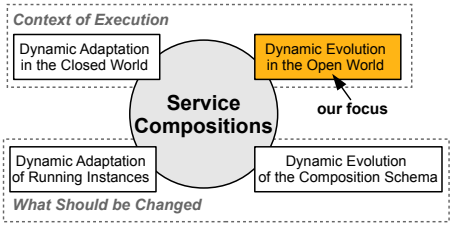

Figure 2. Dimensions of dynamic adaptation and dynamic evolution in service compositions

On the other hand, the two dimensions in the top of Figure 2 are related to the *context of execution*. The dimension of *dynamic adaptation in the closed world* is focused on the closed-world assumption, in which all context events are foreseen at design time. Predefined actions guide adaptations according to known context events (for example, through event-condition-action – ECA – rules). There are several research works that focus on this dimension [6], [7], [8]. However, predefined adaptation actions for known context events are not enough in the open world where several unknown context events can arise (e.g. sudden security attacks). Despite the recognized need for handling unexpected events in self-adaptive systems [14], [15], the dimension of *dynamic evolution in the open world* of service compositions is still an open and challenging research topic. Our work focuses on this dimension.

IV. Facing Uncertainty with Dynamic Evolution

In our previous work [8], we described a model-driven approach to support the dimension of dynamic adaptation of service compositions in the closed world. In order to support the dynamic evolution of service compositions in the open world, we extend this dimension with a *dynamic evolution layer*. Highly abstract *tactics* are the main components to face uncertainty in service compositions that run in the open world. The use of tactics is common in sports, war, or even in daily matters to accomplish an end. For example, the most important *goal* during a battle is to win. However, unknown or unforeseen events, such as surprise assaults, may arise. These events may negatively affect the expected goal. Therefore, it is necessary to choose among a set of tactics to reach the goal (e.g. to escape vs. to do a frontal attack).

We define *tactics* as last resort surviving actions or strategies to preserve the requirements (i.e., goals) that can be negatively impacted by unknown context events [9]. Therefore, tactics can trigger the dynamic evolution of a service composition to preserve its requirements, which were defined at design time[1]. During dynamic evolutions, requirements and tactics need to be known beforehand. Otherwise, it will be impossible to face uncertainty in a controlled way. However, they are not attached to any context event or specific reconfiguration actions (as dynamic adaptations do [8]). Therefore, our approach manages *known unknowns*: tactics are known beforehand, but we do not know to which specific arising unknown context events they will be applied.

In the open world, our approach tries to reduce the impact of unknown context events with a group of tactics. Therefore, the open world can be seen as: ***Open world*** = (∑***unknown context events that can be handled by tactics***) ⋃ (∑***unhandled unknown context events***).

Figure 3 shows the architecture of the dynamic evolution layer to face uncertainty in the open world. This architecture has three building blocks: 1) the **Evolution Planner** constantly looks for unknown context events in the open world. If there is an unknown context event, then it looks for a requirement that can be affected by this event. Afterwards, it looks for a surviving tactic to preserve the requirement; 2) the **Reconfiguration Engine** performs the necessary evolution in two main steps. First, it merges the discovered tactic into a composition model, which abstracts the service composition. This evolved composition model supports the tactic's functionality to preserve the affected requirement. Second, it evolves the WS-BPEL composition schema according to the evolved composition model; and 3) the **Execution Engine** deploys the evolved WS-BPEL composition schema.

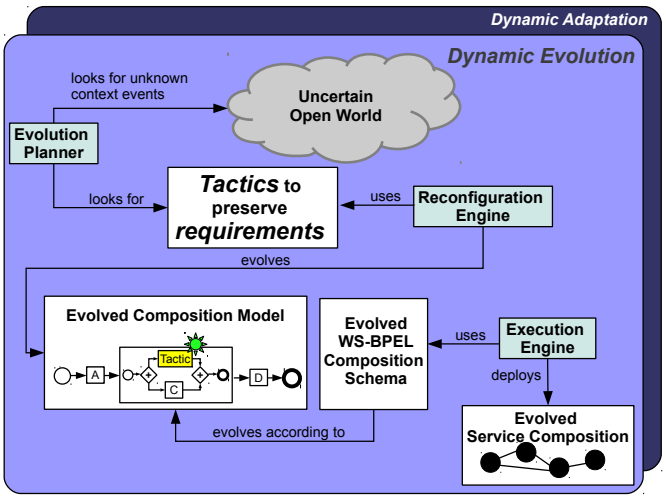

Figure 3. Architecture of the dynamic evolution layer to face uncertainty

[1]There is a main difference between tactics and compensations mechanisms. On one hand, compensation mechanisms try to reverse actions performed in a transaction when failures are faced. On the other hand, tactics try to preserve expected requirements that may be affected by arising problematic unknown context events.

V. Supporting Models for Dynamic Evolution

Our approach requires pieces of knowledge during execution to reach dynamic evolution of service compositions. These pieces are defined as abstract models, which are described as follows (see Figure 4).

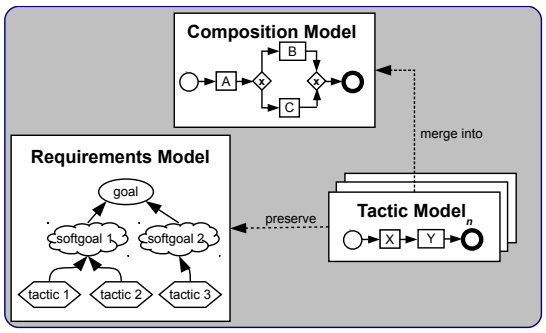

Figure 4. Supporting models for dynamic evolution of service compositions

Composition Model: The composition model abstracts the underlying service composition (e.g. the model in Figure 1). A BPMN model was chosen to represent the elements in a service composition because BPMN is suitable to express sequences and dependencies among Web services and composite services.

Requirements Model: The requirements model is leveraged at runtime to count on the representation of the requirements that the service composition has to preserve at runtime (despite arising unknown context events). Since our running example is particularly interested in keeping non-functional requirements (NFRs) at runtime, the Goal-oriented Requirements Language (GRL) [16] has been used for requirements modeling because it is focused on NFRs. Figure 5 shows the requirements model for our running example. *Softgoals* describe the NFRs to be kept by the service composition in order to reach the top-level *goal*, and *tasks* specify particular surviving tactics to reach softgoals. There is a one-to-many relationship between a requirement (expressed as a softgoal in our case) and tactics that can preserve it. A large set of available generic tactics can be found in related work [17].

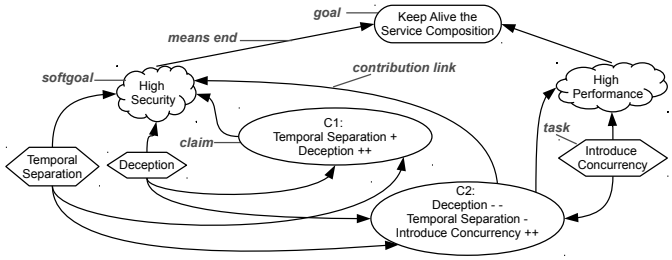

Figure 5. Fragment of the requirements model for the running example

Conflicts may exist between tactics and softgoals. For example, the application of the Deception tactic that preserves the High Security softgoal can negatively impact the High Performance softgoal (more computational resources are required). In order to solve conflicts, we define *claims*, which

indicate assumptions about operationalizations' satisfaction of softgoals [18]. Tactics specified in claims use an ordinal scale, ranging from complete denial (- -) to complete satisfaction (+ +) to express softgoals satisfaction. If a claim states that a tactic operationalization has a negative impact on a softgoal, then another tactic can be tried. For example, Figure 5 shows two claims. At runtime, if an unknown context event affects the High Security softgoal, the Evolution Planner chooses the Deception tactic first according to C1 because it has a better impact on this softgoal. If this tactic does not solve the problem, then the Temporal Separation tactic can be triggered. In a more complex scenario, if an unknown context event affects the High Security and High Performance softgoals at the same time, C2 is checked to decide the best tactic to choose from. In this case, the Temporal Separation and Introduce Concurrency are chosen because they have the most positive impact on these softgoals.

Tactic Models: Tactic models express the tactical functionality to be triggered on the service composition to preserve affected requirements. Therefore, tactic models are causally connected to software (e.g. Web services) that implement the tactics. Tactic models are merged into the composition model at runtime to include the tactical functionality in the evolved service composition. The only merging prerequisite is that these two models conform to the same metamodel. Figure 6 shows a tactic model in our running example. Since the composition model is implemented as a BPMN model, this tactic model is also expressed as a BPMN model to merge it at runtime. There are not limits for the length or depth of model elements used in tactic models. For example, the collapsed Log Intruder's Activities subprocess in Figure 6 contains the BPMN activities that describe the invocation of service operations for writing in the network and data logs.

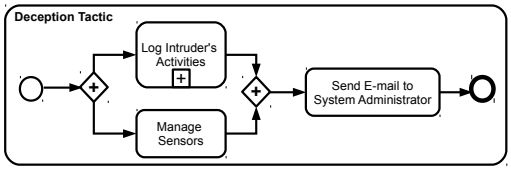

Figure 6. Deception tactic model

VI. Mechanisms for Dynamic Evolution

The models created at design time are used to manage external uncertainty when facing unknown context events. The Evolution Planner and the Reconfiguration Engine are in charge of guiding the dynamic evolution of service compositions. They are described in the following subsections.

A. Evolution Planner

The main objective of the Evolution Planner is to look for surviving tactics to protect the requirements that can be affected by problematic unknown context events. Therefore, the first two steps to trigger a dynamic evolution are as follows:

1. To Observe the Context: In order to collect context information, a context monitor [8] periodically observes the context. It leverages the OWL Web Ontology Language to periodically insert new facts in an ontology that abstracts the context. *Individuals* in this ontology represent service operations. Each individual has *datatype properties* that are used to represent the current context state (e.g. the *isAvailable* datatype property indicates if a service operation is currently available).

2. To Look for Unknown Context Events from the Collected Information: In order to look for unknown context events, the Evolution Planner periodically checks the updated ontology. **An observed context event is considered as *unknown* when there are not predefined context conditions to deal with it**. *Context conditions* are Boolean expressions that work as SLAs [8]. If a context condition is accomplished (i.e., an SLA is violated), then an adaptation is triggered on the service composition to deal with the arising situation (e.g. *UPSShipping, HasResponseTime, > 2,000 ms*). Our approach only observes changes in datatype properties of the ontology. In other words, we do not currently consider discovered service operations in the open world that may affect the structure of the ontology.

In order to face unknown context events, the Evolution Planner carries out the following steps:

1. Search Affected Requirement(s): In order to find the requirement(s) that can be affected by unknown context events, the Evolution Planner uses *forward chaining*. This method evaluates arising context *facts* (i.e., context events) against general *rule premises* in a knowledge base. A key advantage of forward chaining in the open world is that new context events can trigger new inferences. The knowledge base is implemented as a rule file, which includes RDFS rules (e.g. *[rule1: (?f pre:serviceOperation ?a) (?u pre:rapidIncreaseResponseTimeInThreeMin ?f) -> (?u pre:underAttack ?a)]*). For simple service compositions, rules can be obtained from human experts by: 1) collecting empirical data from the current service composition; 2) analyzing collected data to discover the symptoms of problematic situations; and 3) defining general situations in the context that can affect requirements. In complex service compositions, these steps can be extended with methods for generating rules from data (e.g. with heuristics or neural networks).

Figure 7 shows a basic example when the unknown context event *F1* (a fact) is detected. In this case, rule *R1* has a condition that matches this new fact (step 1). Then, the forward chaining method fires the new fact *F2* (step 2). The process continues until the fact *F3* is fired (step 4). *F3* indicates that *the **Barnes & Noble Books service operation** can affect the **High Security softgoal***. This example shows that evolutions are only triggered when requirements are negatively impacted.

2. Search Surviving Tactics: The objective of this step is to discover a tactic to preserve a requirement that can be negatively impacted by an unknown context event. To this end, the Evolution Planner carries out the following steps: 1) it looks for the affected softgoal in the requirements model; 2) it checks the claims associated to the affected softgoal and the

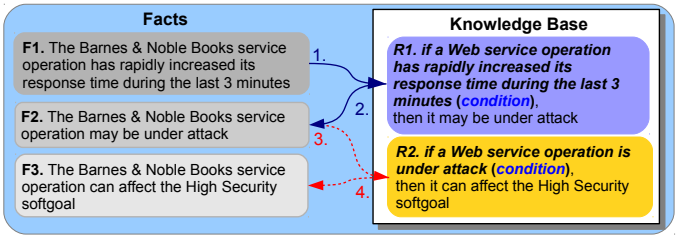

Figure 7. Forward chaining inference example

set of tactics that depend on this softgoal; and 3) according to claims, it chooses the tactic with the most positive impact on the softgoal. For example, when the Evolution Planner finds that the *Barnes & Noble Books service operation can affect the High Security softgoal*, it looks for the High Security softgoal in the requirements model. According to claim C1 in Figure 5, the Evolution Planner chooses the Deception tactic, with the most positive impact on this softgoal.

B. Reconfiguration Engine

The main objective of the Reconfiguration Engine is to evolve the composition model with the tactic that has been found by the Evolution Planner. The Reconfiguration Engine carries out the following steps during evolution:

1. Merge a Tactic Model into the Composition Model: In order to inject the functionality of the discovered tactic into the service composition, it is necessary to: 1) identify a tactic model that describes the tactic to be triggered for preserving an affected requirement; and 2) to merge the required tactic model into the composition model to count on an enriched composition model that guides changes in the service composition. The merging operation was inspired by the *insert process fragment* pattern described in [13].

The set of steps that are carried out in the merging operation are as follows (see Figure 8): 1) the activity (e.g. subprocess or task) describing the service operation that can negatively affect a requirement is put into a new subprocess; 2) the discovered tactic is put into the created subprocess; 3) a parallel relationship is created between the problematic activity and the tactic. As a result, the tactic's functionality will be executed when the problematic service operation is invoked; 4) the sequence flows that come in and go out from the problematic activity are redirected to the created subprocess. If subsequently, an activity that has been preserved with a tactic needs to be removed (i.e., in case of a triggered predefined dynamic adaptation), then the tactic is also removed. Figure 8 shows the evolved composition model after discovering the Barnes & Noble Books service operation can affect the High Security softgoal.

2. Evolve the WS-BPEL Composition Schema: The evolution of the WS-BPEL composition schema is guided by the information contained in the evolved composition model. In our previous work, we used the BABEL Java tool[2] to translate

[2]http://www.bpm.scitech.qut.edu.au/research/projects/oldprojects/babel/tools/: Babel.

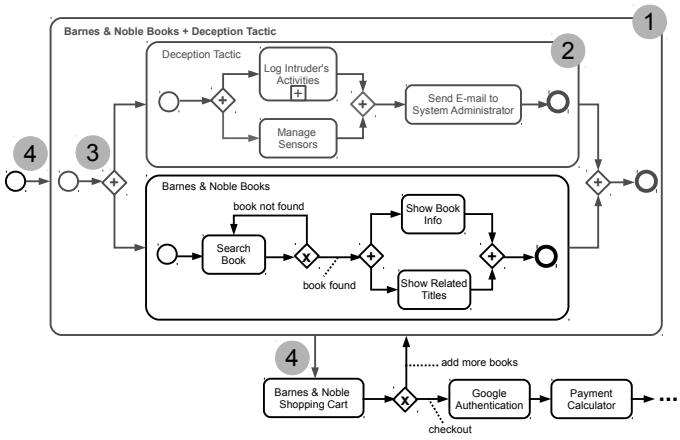

Figure 8. Evolved composition model in our running example

BPMN models into WS-BPEL code [19]. However, we discovered that model-to-model and model-to-text transformations in BABEL are unfeasible at runtime because they take around 95% of the total time required for adjustments.

We propose the following steps to reflect the changes in the evolved composition model into the WS-BPEL composition schema (see Figure 9): 1) the Reconfiguration Engine looks for the tactic that has been added into the composition model; 2) with this information, the Reconfiguration Engine looks for the WS-BPEL code fragment that invokes the tactical functionality. Each tactic model maps to a WS-BPEL code fragment, which is stored in a repository (i.e., a directory). Each code fragment has an associated WSDL, which is used to invoke the tactic's Web service; and 3) the Reconfiguration Engine injects the WS-BPEL code fragment that invokes the tactic into the composition schema. A parallel flow is dynamically created between the code that invokes the affected service operation and the code that invokes the tactic's Web service.

In each evolution, the Reconfiguration Engine puts the evolved composition schema and other required artifacts (e.g. WSDL files) into a deployment directory. This directory is hot deployed by the Execution Engine. Each new directory has a higher version to prevent the Execution Engine from deleting all the running instances with new deployments. New instances run according to the evolved composition schema. Therefore, our approach also covers the dimension of dynamic evolution of the composition schema. Existing approaches, such as Weber et al. [13], offer a solution to migrate running instances to cope with the evolved composition schema. Finally, instead of extending the functionality of the Execution Engine, our approach offers a transparent solution: it can be plugged/unplugged from the the Execution Engine without modifying it.

VII. Prototype and Evaluation

In our prototype, the models were specified in the XML Metadata Interchange (XMI) format to be queried at runtime. The composition model and the set of tactic models were

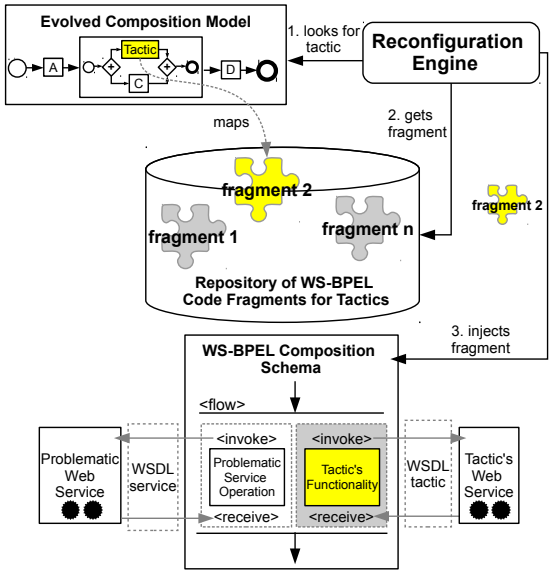

Figure 9. Evolution of the WS-BPEL composition schema

created using the metamodel provided by the Eclipse BPMN Modeler[3]. SALMon [20] inserts new facts into the ontology that represents the context. The Evolution Planner uses Jena 2[4] for forward chaining. The Evolution Planner and the Reconfiguration Engine use the Eclipse Modeling Framework (EMF) Model Query[5] to carry out operations on models. The Reconfiguration Engine is implemented with our Model-based Reconfiguration Engine for Web Services (MoRE-WS) [8]. The Execution Engine was implemented with Apache ODE[6]. Apache ODE was chosen because it offers mature hot-deployment support. Our prototype provides a graphical IDE to facilitate the creation and visualization of models (see Figure 10). A demonstration of our prototype and the models that were used in the running example can be found on our website[7].

We carried out the following set of experiments to evaluate the feasibility of our approach. These experiments were performed on a PC with an Intel Core 2 Duo 2.0 GHz processor and 4 GB RAM with Ubuntu version 10.04 and Kernel Linux version 2.6.32-36-generic. All the Web services ran on the same computer.

1) Searching Affected Requirements: We evaluated the accuracy and performance of the inferences that look for the requirements that can be affected by unknown events. To this end, we purposely injected a set of events that were not predefined at design time to simulate uncertainty in the open world. We simulated some security attacks (e.g. DoS attacks), performance decrease in some service operations (by manually modifying the response times in the execution log), and the unavailability of other operations (by stopping some services). Figure 11 summarizes the results of 16 runs with increasing rules in the knowledge base and unknown events. Our approach found the affected requirements in 83.9% cases. The number of discovered affected requirements is directly proportional to the number of rules. This operation took 57 milliseconds in average.

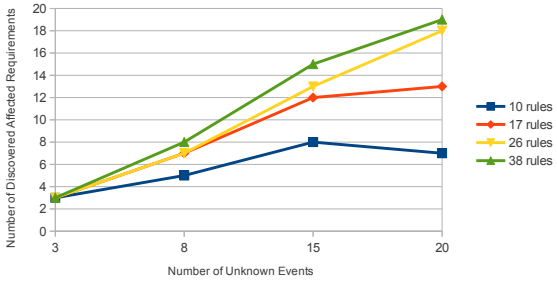

Figure 11. Number of discovered affected requirements

2) Searching Surviving Tactics: We measured the response time when looking for surviving tactics in the requirements model. Since the response time to find a tactic in our running example took seven milliseconds in average, we decided to scale up the number of tactics in the requirements model. For 183 tactics, the response time was 63 milliseconds. In general, the response time of this operation is linear as the number of elements in the requirements model increases (see Figure 12).

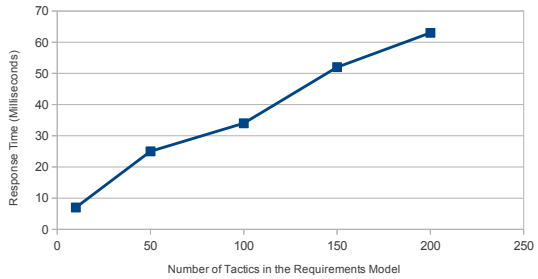

Figure 12. Response time when searching surviving tactics

3) Model-based Evolution: We measured the response time of three key operations that are carried out on models during dynamic evolutions. In order to demonstrate that our approach scales well for large models, we randomly generated large composition models (see Figure 13).

First, the *searching problematic service operation* exhaustively navigates the composition model to look for an activity that describes a problematic service operation. Second, the *adding tactic into the composition model operation* looks for the activity that represents the affected service operation, adds the necessary model elements, and updates the composition model. Therefore, it took longer than the first operation. Third, the *removing tactic from the composition model operation* got a better response time because it only deletes the tactic-related elements and updates the model. Overall, even with a model population of 30,000 elements in the composition model, model operations had a good time response (< 300

[3]http://projects.eclipse.org/projects/soa.bpmnmodeler: BPMN Modeler.
[4]http://incubator.apache.org/jena/: Jena.
[5]http://www.eclipse.org/modeling/emf/: EMF Model Query.
[6]http://ode.apache.org/: Apache ODE.
[7]http://www.harveyalferez.com/dynamicevolutionservcomp/: Prototype.

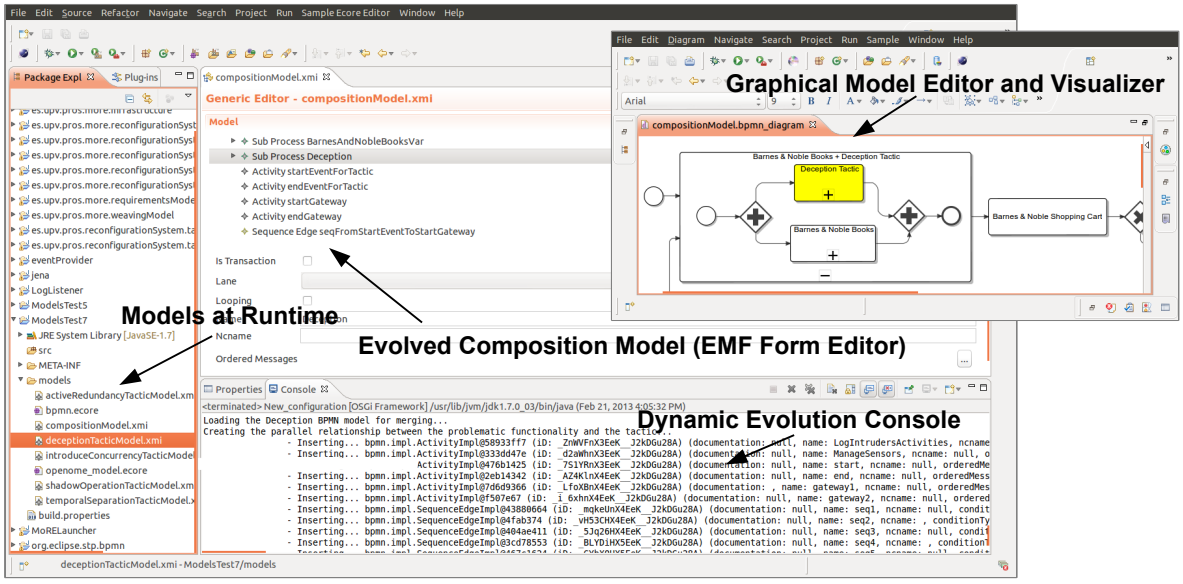

Figure 10. Screenshots of the prototype

milliseconds). It can be considered fast in the domain that we are addressing. Finally, our approach works well under stress circumstances. When several unknown events are achieved in tight time frames, the Reconfiguration Engine executes the required evolutions in the order in which events arrive.

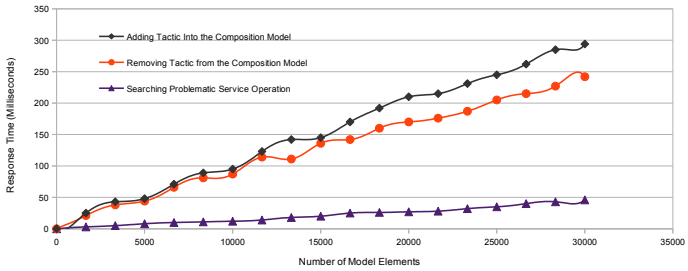

Figure 13. Response time of model operations during dynamic evolutions

4) Discussion: The evaluation demonstrated that model-driven dynamic evolution of service compositions has good performance and scales well. In order to increase the effectiveness of our approach in the open world, the number of rules in the knowledge base, and their related abstract requirements and surviving tactics have to be proportional to the complexity of the context of execution. Although our approach does not solve uncertainty completely, it is an important step towards uncertainty management.

VIII. Related Work

Several research works related to autonomic service compositions have tended to implement variability constructs at the language level to guide dynamic adaptations in the closed world. For example, SCENE [1] extends WS-BPEL with ECA rules that define consequences for conditions to guide the execution of binding and rebinding self-reconfiguration operations. VxBPEL [2] is an adaptation of WS-BPEL that allows variation points, variants, and configurations to be defined for a process in a service-centric system. In [3], monitoring directives are expressed in the Web Service Constraint Language, and recovery strategies, which follow the ECA paradigm, are stated in the Web Service Recovery Language. Also, Aspect-Oriented Programming has been proposed to guide self-adaptive service compositions [4]. However, implementing and managing dynamic adjustments at the language level can become complex as mentioned before. Our solution uses easy-to-understand models at runtime to guide the dynamic evolution of service compositions in the open world.

Traditional attempts to manage context-aware service compositions with models at runtime focus on dynamic adaptation in the closed world, not on dynamic evolution in the open world. For example, DySOA [6] offers components for monitoring and reconfiguring Web service-centric systems using models. QoSMOS [21] combines formal specification of quality-of-service (QoS) requirements, model-based QoS evaluation, monitoring and parameter adaptation of the QoS models, and planning and execution of system adaptation. SASSY [7] is a model-driven framework that provides runtime adaptation of service compositions in response to changing operating conditions.

There are several approaches that deal with modeling variability in service compositions that support BPs [22], [23], [24]. Although these works have inspired ours, they integrate all possible process variants in a single model. It results in large and difficult-to-understand models. On the contrary, we propose to reason about variable tactics separately and to leverage models at runtime to guide dynamic evolutions.

Recently, the community of models at runtime has shown interest on using models during execution to face uncertainty in the open world [25]. However, just a small group of works, which are not focused on service compositions, deal with

uncertainty by means of models at runtime [26], [27], [28]. Moreover, the models that are proposed by these approaches do not evolve at runtime. Therefore, the capacity of reaction to manage unknown context events decreases because the initial models are unable to support them.

IX. Conclusions and Future Work

In this paper, we have described a tool-supported approach that leverages models at runtime to guide the dynamic evolution of context-aware service compositions in the open world. Our approach can be used to manage uncertainty produced by unknown context events. The use of models at runtime has the following benefits: 1) the modeling effort made at design time also provides a rich semantic base for autonomic behavior during execution; 2) since models are causally connected to the underlying service composition, they provide up-to-date information to drive subsequent evolutions; and 3) technological bridges are avoided because the model representations that are used at design time are kept at runtime. An evaluation demonstrated that model-driven dynamic evolution of service compositions has good performance and scales well. As future work, we are going to use Constraint Programming to verify the evolved models and check that generated configurations respect the constraints imposed by the models. Also, we are going to verify that the introduction of tactics does not negatively affect other expected goals. In addition, we are going to propose a methodology to build and collect generic tactics.

Acknowledgment

This work has been developed with the support of MICINN under the project everyWare TIN2010-18011 and co-financed with ERDF.

References

[1] M. Colombo, E. Di Nitto, and M. Mauri, "SCENE: A service composition execution environment supporting dynamic changes disciplined through rules," in *Service-Oriented Computing – ICSOC 2006*, ser. Lecture Notes in Computer Science, A. Dan and W. Lamersdorf, Eds. Springer Berlin / Heidelberg, 2006, vol. 4294, pp. 191–202.

[2] M. Koning, C.-a. Sun, M. Sinnema, and P. Avgeriou, "VxBPEL: Supporting variability for web services in BPEL," *Inf. Softw. Technol.*, vol. 51, pp. 258–269, February 2009.

[3] L. Baresi and S. Guinea, "Self-supervising BPEL processes," *IEEE Trans. Softw. Eng.*, vol. 37, pp. 247–263, March 2011.

[4] M. Sonntag and D. Karastoyanova, "Compensation of adapted service orchestration logic in BPEL'n'aspects," in *Proceedings of the 9th International Conference on Business Process Management (BPM 2011), Clermont-Ferrand, France, 2011*. Springer-Verlag, August 2011, pp. 1–16.

[5] L. Baresi, E. Di Nitto, and C. Ghezzi, "Toward open-world software: Issue and challenges," *Computer*, vol. 39, pp. 36–43, October 2006.

[6] I. Bosloper, J. Siljee, J. Nijhuis, and D. Hammer, "Creating self-adaptive service systems with DySOA," in *Proceedings of the Third European Conference on Web Services*, ser. ECOWS '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 95–104.

[7] D. Menasce, H. Gomaa, S. Malek, and J. Sousa, "SASSY: A framework for self-architecting service-oriented systems," *IEEE Software*, vol. 28, pp. 78–85, 2011.

[8] G. H. Alférez and V. Pelechano, "Context-aware autonomous web services in software product lines," in *Proceedings of the 2011 15th International Software Product Line Conference*, ser. SPLC '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 100–109.

[9] ——, "Dynamic evolution of context-aware systems with models at runtime," in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, R. France, J. Kazmeier, R. Breu, and C. Atkinson, Eds. Springer Berlin / Heidelberg, 2012, vol. 7590, pp. 70–86.

[10] G. Blair, N. Bencomo, and R. B. France, "Models@ run.time," *Computer*, vol. 42, pp. 22–27, October 2009.

[11] N. Esfahani, E. Kouroshfar, and S. Malek, "Taming uncertainty in self-adaptive software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 234–244.

[12] A. Marconi, M. Pistore, A. Sirbu, H. Eberle, F. Leymann, and T. Unger, "Enabling adaptation of pervasive flows: Built-in contextual adaptation," in *Proceedings of the 7th International Joint Conference on Service-Oriented Computing*, ser. ICSOC-ServiceWave '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 445–454.

[13] B. Weber, M. Reichert, and S. Rinderle-Ma, "Change patterns and change support features - enhancing flexibility in process-aware information systems," *Data Knowl. Eng.*, vol. 66, pp. 438–466, September 2008.

[14] R. Calinescu, C. Ghezzi, M. Kwiatkowska, and R. Mirandola, "Self-adaptive software needs quantitative verification at runtime," *Commun. ACM*, vol. 55, no. 9, pp. 69–77, Sep. 2012.

[15] B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, Eds., *Software Engineering for Self-Adaptive Systems*. Berlin, Heidelberg: Springer-Verlag, 2009.

[16] L. Liu and E. Yu, "Designing information systems in social context: a goal and scenario modelling approach," *Inf. Syst.*, vol. 29, pp. 187–203, April 2004.

[17] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed. Addison-Wesley Professional, 2012.

[18] P. Sawyer, R. Mazo, D. Diaz, C. Salinesi, and D. Hughes, "Using constraint programming to manage configurations in self-adaptive systems," *Computer*, vol. 45, no. 10, pp. 56 –63, oct. 2012.

[19] V. Torres, P. Giner, and V. Pelechano, "Developing BP-driven web applications through the use of MDE techniques," *Software and Systems Modeling*, vol. 11, pp. 609–631, 2012.

[20] D. Ameller and X. Franch, "Service level agreement monitor (SALMon)," in *Proceedings of the Seventh International Conference on Composition-Based Software Systems (ICCBSS 2008)*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 224–227.

[21] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli, "Dynamic QoS management and optimization in service-based systems," *IEEE Transactions on Software Engineering*, vol. 37, pp. 387–409, 2011.

[22] F. Puhlmann, A. Schnieders, J. Weiland, and M. Weske, "Variability Mechanisms for Process Models," Tech. Rep., June 2005. [Online]. Available: http://frapu.de/pdf/PESOA_TR_17-2005.pdf

[23] M. Rosemann and W. M. P. Van der Aalst, "A configurable reference modelling language," *Inf. Syst.*, vol. 32, no. 1, pp. 1–23, Mar. 2007.

[24] F. Gottschalk, W. M. P. van der Aalst, M. H. Jansen-Vullers, and M. L. Rosa, "Configurable workflow models," *Int. J. Cooperative Inf. Syst.*, vol. 17, no. 2, pp. 177–221, 2008.

[25] U. Aßmann, N. Bencomo, B. H. C. Cheng, and R. B. France, "Models@run.time (dagstuhl seminar 11481)," *Dagstuhl Reports*, vol. 1, no. 11, pp. 91–123, 2011.

[26] K. Welsh, P. Sawyer, and N. Bencomo, "Towards requirements aware systems: Run-time resolution of design-time assumptions," in *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, nov 2011, pp. 560 –563.

[27] B. H. Cheng, P. Sawyer, N. Bencomo, and J. Whittle, "A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty," in *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 468–483.

[28] H. J. Goldsby and B. H. Cheng, "Automatically generating behavioral models of adaptive systems to address uncertainty," in *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, ser. MoDELS '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 568–583.