Proceedings of the 50th Hawaii International Conference on System Sciences | 2017

Adaptive service composition based on runtime verification of formal properties

Glaucia M. M. Campos
State University of Rio Grande do Norte
Federal University of Pernambuco
Center of Informatics
Recife, Brazil
gmmc@cin.ufpe.br

Nelson Souto Rosa
Federal University of Pernambuco
Center of Informatics
Recife, Brazil
nsr@cin.ufpe.br

Luís Ferreira Pires
University of Twente
Faculty of Electrical Engineering,
Mathematics & Computer Science
Enschede, The Netherlands
l.ferreirapires@utwente.nl

***Abstract*—Service-Oriented Computing (SOC) has been used in business environments in order to integrate heterogeneous systems. The dynamic nature of these environments causes changes in the application requirements. As a result, service composition must be flexible, dynamic and adaptive, which motivate the need to ensure the service composition behavior at runtime. The development of adaptive service compositions is still an opportunity due to the complexity of dealing with adaptation issues, for example, how to provide runtime verification and automatic adaptation. Formal description techniques can be used to detect runtime undesirable behaviors that help in adaptation process. However, formal techniques have been used only at design-time. In this paper, we propose an adaptive service composition approach based on the lightweight use of formal methods. The aim is detecting undesirable behaviors in the execution trace. Once an undesirable behavior is detected during the execution of a service composition, our approach triggers an adequate reconfiguration plan for the problem at runtime. In order to evaluate the effectiveness of the proposal, we illustrate it with a running example.**

***Keywords*-service composition; formalisms; runtime verification; adaptation**

I. Introduction

Service-Oriented Computing [1] has been introduced in business environments to cope with heterogeneous systems integration. SOC is based on the concept of service and it brings benefits as platform and language interoperability, low coupling and functionality reuse. However, the full potential of SOC can be only fully exploited with the possibility of composing existing services that integrates different capabilities across companies′ boundaries.

Services run in business environments that continuously change, and they are also required to change in order to respond to the demands of these dynamic environments. These changes can appear as the deployment of new instances of a particular service that removes the undesirable behavior, adds new functionalities or optimizes its quality characteristics in terms of performance, security or availability. When these demands emerge from the environment, the service composition should be able to continue functioning despite the changes in the service it uses.

As a consequence of the aforementioned dynamism, service composition must become more flexible and it must be able to detect and react to changes. Because of these challenges, the adaptation is an important element for the proper functioning of the service composition. Therefore, adaptation [2] can be defined as a process of modifying applications in order to satisfy their new requirements and to meet the demands imposed by the environment on the basis of reconfiguration plans.

The development of an adaptive service composition is still an opportunity due to the complexity of dealing with adaptation issues, such as how to provide runtime verification, how to implement adaptation mechanisms, when to carry out the adaptation and where to perform the adaptation. It is also difficult to identify transient undesirable behaviors and repair the service composition on time through actions that do not require its suspension. In order to provide the verification of the service composition at runtime, formal description techniques can be used by checking the behavioral properties. However, most approaches on verification and validation services composition using formal techniques are only focused on the service composition analysis at design time. Then, it is necessary to develop new solutions that dynamically reconfigure service composition with the aid of formal techniques.

Considering this scenario, this paper presents an approach for building adaptive service compositions based on the lightweight use of formal techniques to detect undesirable behaviors in their execution traces. Initially, we instrument the interactions among services in order to record the execution trace. We describe the functional requirements of the service composition using temporal logic, which allows us to check the behavior consistency according to the execution trace. Once an undesirable behavior (deviation) is detected during the execution of a service composition, our approach triggers an adequate reconfiguration plan for the problem at runtime. These plans can range from simply trying to reinvoke the service up to replacing it by another equivalent service.

The main contributions of this paper include the definition of an approach where existing service compositions are analyzed by means of execution traces, the lightweight use of tools that support formal behavioral properties verification at runtime, the adaptation of the service composition at runtime through actions that do not require its suspension and

URI: http://hdl.handle.net/10125/41750
ISBN: 978-0-9981331-0-2
CC-BY-NC-ND

without the human intervention and the use of hierarchical reconfiguration plans. The approach is illustrated with a running example (Portuguese dictionary service composition) and it assesses the impact of system adaptation upon its performance.

This paper is further structured as follows. Section II shows the motivation of the developed approach. Section III describes the approach to adaptive service compositions. In IV, we present the formalization of this approach. Section V presents the implementation of the proposed approach. The running example is presented in Section VI. The results of the proposal evaluation are described in Section VII. Section VIII shows the related works. Finally, Section IX introduces our conclusions and some future works.

II. Motivation

A service composition has been recursively defined as an aggregation of services that generates a new service by providing more elaborated functionalities. It integrates different capabilities across companies′ boundaries. However, there is no guarantee that services will behave as expected due to the dynamism of the business environment. Then, we need to ensure the functional requirements of service composition.

According to the service composition life cycle [1], the functionality of service composition can be verified at design-time or runtime. At design-time, the method ensures that the desirable requirements can be achieved without the system execution, whereas runtime verification concentrates on the service composition behavior when it is on-the fly [3].

Once service composition becomes operational, its progress needs to be frequently monitored in order to ensure functional competencies and desirable quality levels (e.g. performance, security and availability) by runtime verification methods. The collected data correspond to the interactions among services and they represent the behavior of service composition. Once collected, the data must be analyzed according to the adopted verification method. These methods are classified based on system validation time: off-line mode and on-line mode. In an off-line mode, although the required data are collected when the system is on-the-fly, their analysis is performed when the system execution exceeds the time of logged events. In an on-line mode, both data acquisition and analysis are carried out in a timely manner when the system is running.

The result of runtime verification indicates whether the service composition execution satisfies or violates a given desirable property. When a violation is detected, verification methods must be followed by actions that allow the dynamic adaptation of service composition to take it to a safe state. Adaptation has been implemented in different moments: reactive, pro-active and post-morten. Reactive adaptation handles faults reported during the execution of a composition instance, pro-active adaptation modifies a service composition before a deviation can be detected and post-morten adaptation characterizes a significant gap between the detection of the undesirable property and a performed modification.

Adaptation can be achieved through different mechanisms that define how a particular plan is performed, when there is a wide range of available options. Although using formal methods as a plan to implement an adaptation is not mandatory, they provide the underpinnings to model service composition and describe functional properties that are fundamental for the service composition validation process. However, the adoption of formal techniques is still a hard task for developers due to its cost and the degree of specialization needed for its application. In the adaptive service composition context, the degree of formalization may vary, reducing the need for accuracy in certain development tasks, but it also benefits the correctness of development activities. These proposals have been commonly called lightweight formal methods [4].

III. Proposal

We have developed an approach that uses formal methods, in the lightweight way, to manage the service composition adaptation process. It adapts the service composition by checking the conformance between its execution trace and a set of the desirable behavioral properties described in a temporal logic. For instance, assuming that a service composition begins presenting an undesirable behavior (e.g., service does not respond to invocations or delivers messages out of order), our approach detects these problems and activates a reconfiguration plan for this scenario.

Figure 1 shows a general overview of the adaptive service composition model. A service composition is executed by the execution engine while it is monitored to collect its execution trace (events) (1). Meanwhile, the desirable behavioral properties that must be satisfied by service composition during its execution are defined in the formal language (2). Once collected, our approach takes responsibility by the conformity process of checking behavioral properties according to execution trace (3). When an undesirable property is detected, the simplest action would be sending a message to the user. However, the adaptation process (4) allows a reconfiguration plan (e.g. by replacing a service) to be activated without completely stopping the service composition. Once the verification process detects undesirable behaviors, the service composition is reconfigured and the verification process is repeated according to the reconfiguration plan, for instance, every minute. The same plan also defines a time interval for carrying out the verification process when undesirable behaviors are not detected, for example, every three minutes. Regarding this work, we defined those values after performing a set of service composition simulations with different time intervals for the verification process.

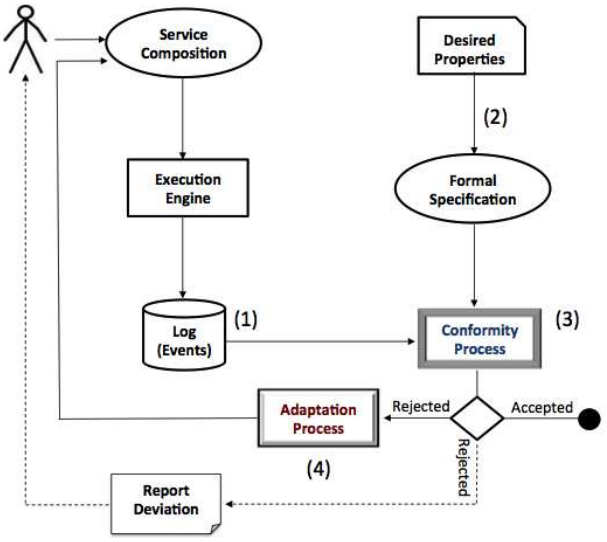

Figure 1. Adaptive Service Composition Model

Considering the concepts related to development, monitoring, verifying and adaptation of service-based applications defined by Hielscher [5] and Papazoglou [1], we can classify our solution as follows:

- the service composition adaptation is performed only after identifying the occurrence of behavioral deviations (reactive adaptation);
- the monitoring process is performed in parallel with the execution of the monitoring subject (asynchronous execution process);
- service composition is represented as a process flow that specifies the order in which the interactions should be executed (service orchestration);
- adaptation plans are pre-defined and associated with the requirement to be adapted (static selection);
- runtime verification is based on execution traces and temporal logic properties;
- services are replaced in order to remove the behavior that is not desirable in a service composition (corrective adaptation);
- the adaptation results are observed only in the following executions of the service composition;
- the execution of adaptation actions is performed at runtime and it does not require any human involvement.

IV. Formalization

In this section, the set of components that makes up our approach is formally described. This formalization has been defined based on works developed by [6] and [7].

Initially, given a service composition *SC*, its execution trace *Tr* and a set of desirable behavioral properties *P*, then *Tr* must satisfies *P* ($Tr \vdash P$). Otherwise, if *Tr* does not satisfy *P* ($Tr \dashv P$), *SC* must be reconfigured at runtime in accordance with a reconfiguration plan *R*, which consists in a set of actions μ performed by the approach.

A. Basic definitions

Firstly,, we define our perception about important concepts of the Service-Oriented Computing, such as *interface component*, *service component* and *service composition*.

Definition 1. (Interface Component). An interface *In* describes the set of operations and parameters that expresses the functionality of the service, where:

- $\forall x \in In$, we define $x = <op, I, O>$, being *op, I, O* correspond to the operations, inputs and outputs respectively;
- For each operation, there is a set of inputs and outputs, i.e., $I = <\delta_1, ..., \delta_m>$ *and* $O = <\beta_1, ..., \beta_n>$, where each of these elements takes the form $<name> : <type>$.

The signatures of these operations are not relevant for this paper, because we are interested in the service behavior, which corresponds to the exchange of messages performed by the service composition.

Definition 2. (Service Component). A service component is represented as a triple $S = < F, In, A >$ where *F, In, A* stand for functions, interfaces and actions respectively:

- *F* describes the functionality of the service;
- *In* is the component interface;
- *A* is the sequence of possible invocations for the component interface.

Other information can be also represented in a service component. For example, a quadruple $S = < F, In, A, P >$, where *P* corresponds to the non-functional properties such as availability, delay, cost and so forth. However, as *P* does not play a significant role in the service composition behavior of the proposed approach, we only consider the triple $< F, In, A >$ for conformance checking.

Definition 3. (Service Composition Component). A service composition component is a triple $SC = < Ns, Rb, B >$, where:

- $Ns = S1, S2, ..., Sn$ is a set of service components that make up the service composition and can be reconfigurable at runtime;
- *Rb* is the binary relation in $Ns \leftrightarrow Rb \subset Ns \; X \; Ns$, which defines how the services are connected. Being $Rb = \{(x, y) \mid x, y \in Ns\}$ and $Ns \neq \emptyset$, when the ordered pair (x, y) belongs to the *Rb* relation, we can write $xRby \leftrightarrow (x, y) \in Rb$;
- *B* is the sequence of actions that can be observed in all the interface components of the service composition.

B. Tracing

Tracing is an important element of this approach because it refers to the content generated by service composition while it is running. Regarding this approach, the execution

trace is produced through the recorded observable actions which represent the service composition behavior.

Definition 4 (Execution Trace). Composition execution trace *Tr* is a set of actions sequences performed by the service composition, where $Tr \subseteq \varphi \cup \epsilon \cup \theta$

- φ is a set of observable actions sequences that can happen in all the interface components of the service composition;
- ϵ is a set of actions sequences that came up due to errors;
- θ is an empty sequence;
- All sequences can be formally defined by a function $\delta : I \rightarrow Tr$, where I is the integer interval $[0, n]$ for some $n \in \mathbb{N}$.

We are interested in a restricted set of operations that the service composition can perform. These operations influence the truth-value of properties because they generate actions that must belong to the service composition vocabulary, namely Σ. Therefore, each set of observable actions sequences φ must be always contained in Σ, i.e., $\forall \varphi_i$ *(i=0, ..., n)* $\in \varphi$, then $\varphi_i \subseteq \Sigma$.

C. Checking Process

The checking process consists in verifying the service composition behavior based on its execution trace and a set of desirable temporal properties formally described. The result corresponds to a logical value that defines if the property have been satisfied ($Tr \vdash P$) or not ($Tr \dashv P$) by the service composition.

Considering an execution sequence φ_i *(i=0, ..., n)* $\in \varphi$, we can define that φ_i satisfies a property Φ_j (j=0, ..., n) $\in P$, when $\varphi_i \in \Sigma \wedge (\Phi_j \rightarrow \varphi_i)$.

In this paper, we are interested in properties that can be verified at runtime. For this purpose, it was adopted the set of property patterns specifications defined in [8] was adopted. They are:

- **Absence.** A given action does not occur within a scope
- **Existence.** A given action must occur within a scope
- **Bounded existence.** A given action must occur K times within a scope
- **Universality.** A given action must occur throughout a scope
- **Precedence.** A given action M must always be preceded by an action N within a scope
- **Response.** A given action M must be always followed by an action N within a scope

From these property patterns and the regular alternation-free Mu-Calculus [9], we define a set of formal properties as described in Table 1. Given a service composition execution trace Tr, the actions sequence φ_i (i=0, ..., n) = $\pi 1$, $\pi 2$, $\pi 3$ $\in Tr$. For a better understanding of the table, characters *T* and *F* must be read like *True* and *False*, respectively.

D. Adaptation Process

Once an undesirable behavior is notified through the verification of a temporal property Φ_i (i=0, ..., n) on the execution trace, several different kinds of reconfiguration plans τ_j (j=0, ..., n) $\in$ R can be triggered at runtime in order to reach the goals of the service composition, regardless of the errors. In other words, there is a specific reconfiguration plan for each undesirable behavior that has been identified:

$$\varphi_1 \dashv \Phi_1 \rightarrow \tau_1$$
$$\varphi_2 \dashv \Phi_2 \rightarrow \tau_2$$
$$\cdots$$
$$\varphi_n \dashv \Phi_n \rightarrow \tau_n$$

Different reconfiguration plans for service-oriented scenarios have been identified in the literature [10] [2], being the following ones the most widely-used actions:

- **Retry.** Executing the same invocation with identical parameters and contracts, according to the number of configured attempts for each service. This is a possible solution when the fault is transient;
- **Replace.** Trying to select another service with similar behavioral characteristics because it is necessary to guarantee the same functional properties;
- **Reorganize.** Creating an on-the-fly composition from a set of available services that can provide the same behavior of a faulty interaction. This is necessary when no alternative matching service can be found.

The use of these reconfiguration plans can be hierarchically organized. For example, the reconfiguration component can start the process from a simple action, like retrying the same service. Then, if it is not possible to reach the goals of the composition, the component can choose another reconfiguration plan, such as replace, and afterwards reorganize. Choosing the reconfiguration plan to recover the service composition is not decided in an ad-hoc manner. This choice depends on the undesirable composition behavior which was detected during the verification process. The hierarchical execution mode of the reconfiguration plans is defined by the reconfiguration specialist who decides the best practices for a specific service composition considering the hierarchical organization of the plans.

Definition 5 (Reconfiguration Plans). A reconfiguration plan can be defined as a function τ: Rb X Rb $\rightarrow$ Rb', where Rb, Rb' are relations that define how the services have been connected.

We need to understand a set of simpler plans (e.g. add and remove plans) which, when combined among them, must represent the plans defined for service composition scenarios.

We need to understand a set of plans simpler (e.g. add and remove plans) that when combined to each other, they must represent the plans defined to service composition scenarios.

Table I
PROPERTIES PATTERNS IN MU-CALCULUS

Description	Formula
$\pi 1$ does not occur	[T* . "$\pi 1$"] F
$\pi 1$ must occurs throughout the sequence	[T* . ¬"$\pi 1$"] F
$\pi 1$ does not occur before $\pi 2$	[(¬"$\pi 1$")* . "$\pi 2$" . (¬ "$\pi 1$")* . "$\pi 1$"] F
$\pi 2$ must occurs before $\pi 3$	[(¬ "$\pi 2$")* . $\pi 3$] F
$\pi 1$ does not occur after $\pi 2$ until $\pi 3$	[T* . "$\pi 2$" . (¬ "$\pi 3$")* . "$\pi 1$"] F
$\pi 1$ must occurs between $\pi 2$ and $\pi 3$	[T* . "$\pi 2$" . (¬ ("$\pi 1$" or "$\pi 3$"))* . "$\pi 3$"] F
No deadlock sequence is found	[T*] < T > T

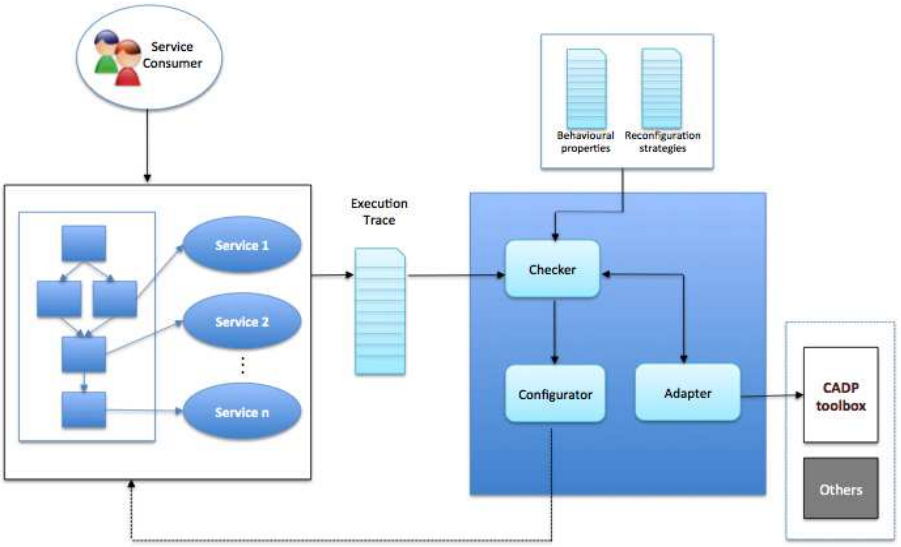

Figure 2. A general overview of the implemented approach

- **Add.** τ_1 (Rb, $\{(S_i, S_n)\}$) = Rb $\cup$ $\{(S_i, S_n)\}$, where $S_i \in$ Ns and S_n is a new service to be added to the service composition;
- **Remove.** τ_2 (Rb, $\{(S_i, S_r)\}$) = Rb $\setminus$ $\{(S_i, S_r)\}$, where S_i and $S_r \in$ Ns and S_r is the service to be removed from the service composition;
- **Retry.** τ_3 (Rb, $\{(S_i, S_r), (S_i, S_n)\}$) = τ_2 (Rb, $\{(S_i, S_r)\}$) $\oplus$ τ_1 (Rb, $\{(S_i, S_r)\}$), where S_i, $S_r \in$ Ns and $S_r = S_n$ is the service to be removed and added again;
- **Replace.** τ_4 (Rb, $\{(S_i, S_r), (S_i, S_n)\}$) = τ_2 (Rb, $\{(S_i, S_r)\}$) $\oplus$ τ_1 (Rb, $\{(S_i, S_n)\}$), where S_i, $S_r \in$ Ns, S_r is the service to be removed and S_n is the new service to be added;
- **Reorganize.** τ_5(Rb, $\{(S_i,S_n)\}$) = Rb $\cup$ $\{(S_i,\{(S_{n1},S_{n2})\})\}$, where $S_i \in$ Ns and $\{(S_{n1},S_{n2})\}$ is a new service composition that can provide the same behavior of a faulty interaction.

V. IMPLEMENTATION

Our approach was implemented in the Java language and it used CADP Toolbox *(Construction and Analysis of Distributed Processes)* [11] to check the execution trace. The traces are text files in CADPs sequence format which can be viewed as an LTS *(Labelled Transition System)*. The SEQ.OPEN tool connects the trace to the model checker where it can be analyzed. This tool works on-the-fly, i.e., without storing in memory the entire content *filename*.seq. Figure 2 shows a general overview of the implemented approach.

The trace is obtained by instrumentation of the service composition. It is responsible for logging actions executed by service composition (i.e. interactions among services). Before the checking process is activated, it is necessary to modify the trace file according to the file format supported by verification tool. In this paper, a ***CADP Adapter*** that modifies the log in accordance with the sequence format was developed, i.e., *tracefilename*.seq. Once the trace is formatted, the ***Checker*** invokes the model checker of the CADP Toolbox to check the behavioral properties, which are described in Mu-Calculus, a model checking language.

The answer from CADP Toolbox is evaluated by Checker module. Once an undesirable behavior is notified, a reconfiguration plan must be triggered in the attempt to reach the desirable behavior of the service composition. Then, Checker

asks the Configurator module to execute the reconfiguration plan in accordance with the unsatisfied property.

Different reconfiguration plans have been identified for service-oriented scenarios and some of them were implemented in the proposed approach:

```
public interface IReconfigurationPlans {
    public void addService (int idx, ServiceCall newServ);
    public void removeService(ServiceCall oldServ);
    public void retryService(ServiceCall new_oldServ);
    public void replaceService(ServiceCall oldServ,
                               ServiceCall newServ);
}
```

Although CADP has been adopted as the formal toolbox, other behavioral verification tools can be added to our solution, for example, ProM framework (PROM2) ***(Process Mining Toolkit)*** [12] through LTL Model Checker. It has been used to verify behavioral properties in business processes at design time, although the business process activities can be performed by services. By using CADP Toolbox, we can choose to check different properties by trace file, such as deadlocks, temporal formulas, generating tests and evaluating performance. In this paper, the reconfiguration process is based only on the verification of the temporal formulas, i.e., Mu-Calculus.

VI. Running Example

The example presented in this section is used to introduce the behavioral problems that can arise in service composition scenarios and the solutions we propose to cope with them. In this example, a client needs the meaning of a Portuguese word. For this purpose, a service composition is initially implemented, as shown in Figure 3. We implemented a Portuguese dictionary service P, which returns the meaning of the word in Portuguese. It is a service composition using three different services: Portuguese-English Translator (PEtranslator), English Dictionary (ENGdictionary) and English-Portuguese Translator (EPtranslator). Initially, the given word is translated from Portuguese to English and its meaning is found in the English Dictionary. Then, the result is obtained from English to Portuguese. We implemented and deployed all the services that take part of the service composition. They were implemented in Java by the Eclipse tool and deployed in the Tomcat web server.

Since the Portuguese dictionary service composition is running, we extract the execution trace described in Figure 4(a). An example of the Service Composition information exchange between client (i.e. consumer of the service) and service composition is not mentioned here. Different kinds of behaviors can be analyzed in this service composition. Regarding the aims of this paper, we consider as undesirable behavior the lack of messages among services in the execution trace. For example, Figure 4(b) presents a scenario where the Portuguese English Translator service does not respond to the request of service composition. There are

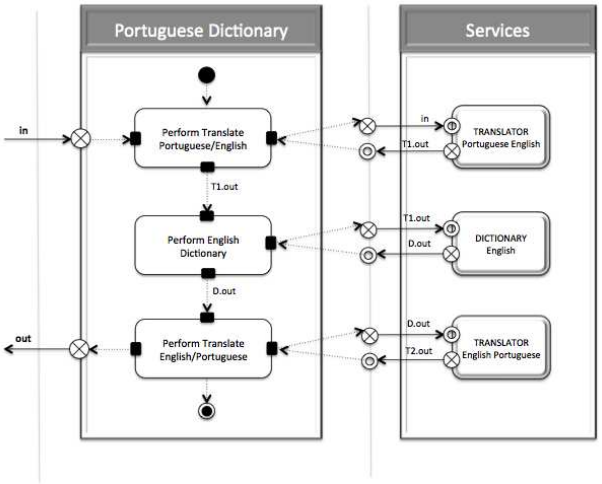

Figure 3. Portuguese Dictionary - An example of the Service Composition

two possible reasons for this situation to occur: service can go down or it does not have the word to be translated. After realizing the lack of message, our approach triggers an adequate reconfiguration plan to solve the problem.

We implemented a reconfiguration hierarchical strategy of the plans. Thus, when the absence of a message is detected, firstly the service is re-invoked to avoid a change of services due to transient undesirable behavior (e.g. temporary server breakdown or messages that have been lost in the net). After the reinvocation, if the undesirable behavior continues being detected, the service that generated this behavior must be replaced by another service with similar functionality. Considering that the services replacement is an expensive process, a service change can be avoided if the undesirable behavior is transient.

Once the services change has been carried out, the service composition execution trace continues being analyzed through the CADP Toolbox model checker. In the trace, other undesirable behaviors through the services invocation can be detected. The same service that replaces the service presenting undesirable behavior can also generate the same behavior (Figure 4(b)). Once the reconfiguration is applied, the execution trace must be according to what is shown in Figure 4 (a).

Although the service composition was implemented in Java language and it uses the Java runtime environment as execution engine, it could have been implemented in any language since the execution trace was produced in the format accepted by our approach.

VII. Experimental Evaluation

The experimental evaluation of this approach was divided in two main steps. Initially, we proved that the adaptation process functioned well and it changed the service composition behavior. Afterwards, we measured the time spent by the

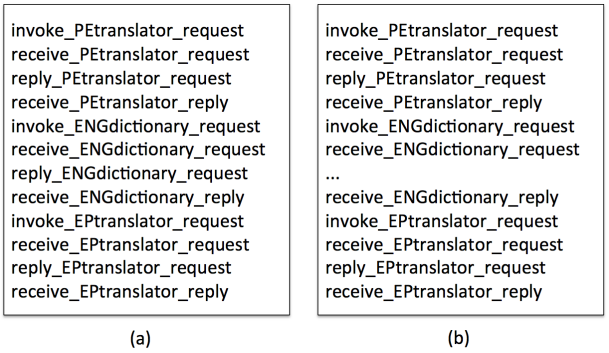

Figure 4. Execution trace of the service composition: (a) desirable execution trace; (b) deviated execution trace

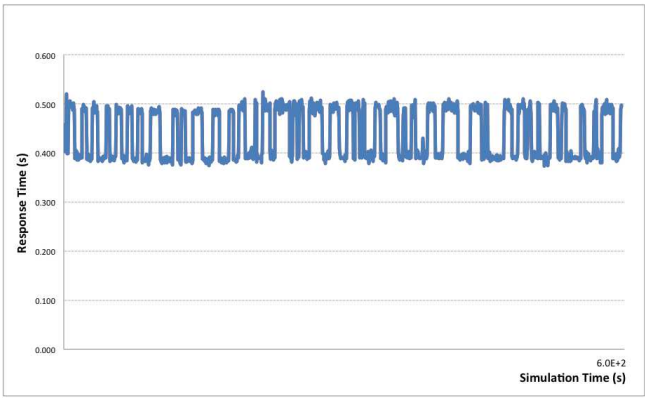

Figure 5. Behavior of the response time when reconfigurations are triggered

approach to realize an undesirable behavior and also perform the adequate adaptation. We also evaluated the overhead solution; in other words, the average time spent on the adaptation activation process.

The experiment was executed in a Mac Book Pro (OS X, version 10.10.1, processor 2.5 GHz Intel Core i5, 8 GB of RAM), where a client performed invocations to a service composition during 6.0E+5ms, and whose service time was set to 600ms. In the meantime, the service composition was invoked 1368 times by the client, and 1137 out of them were correctly processed by service composition. Considering the time spent between the detection of an undesirable behavior and its adaptation, only 166 reconfigurations were performed.

The number of reconfigurations has not been parameterized; random undesirable behaviors are injected in the service composition during the simulation time and they specify the necessary reconfigurations. The number of invocations has also not been parameterized. This number is defined according to simulation time configured at design time. The replacement of the service does not guarantee that the new service can behave appropriately. Then, it is possible that a new reconfiguration should be carried out soon after a first reconfiguration.

A. Adaptation process

In order to show that the adaptation process works well, we initially implemented two *Portuguese-English Translator* services, where *PE-Translator1* service (300ms) takes more time to process a requisition and it can be replaced by *PETranslator2* service (200ms), which is faster. Figure 5 presents the response time of each successful invocation. Also, it shows the behavior of the response time while the adaptations were triggered. The response time corresponds to the time interval between the invocation of the client and the reception of the response.

The figure shows that the response time of the successful invocations occur within a small interval, approximately 0.5 seconds for the slower service and 0.39 seconds for the faster one. When there are some divergent values for the execution of the same service, it means that some reinvocations were performed for these cases; therefore, the time is different from the other invocations. The time response behavior proves that the adaptation has been carried out. When the service that takes the longest time to perform is chosen, the response time is relatively longer (0.5 seconds). At some moment, this time decreases and remains the same during a very small time interval, proving that the service that takes less time to respond to the requisition is being performed at that moment. Figure 5 represents the relationship between the simulation time and the response time of each client (service).

B. Performance evaluation

We are also interested in demonstrating the impact that the adaptation process has upon the service composition performance. For this purpose, we chose two different performance aspects: the overhead caused by adaptation process in the service composition and the time spent from the beginning of an undesirable behavior until the end of the adaptation that corrects this behavior. Then, we considered the average response time (see Section VII - A) as well as the reconfiguration time for each undesirable behavior as performance metrics.

The average response time was calculated for scenarios with the disabled and enabled reconfiguration mechanism. Therefore, we can evaluate the overhead caused by adaptation process in the service composition. According to the collected data, there is an overhead caused by reconfiguration mechanism, even because an additional processing is being carried out in the service composition when undesirable behaviors are detected. Figure 6 shows that there is an overhead in the response time when the reconfiguration mechanism is enabled. However, this overhead is considered low when compared to the time spent with

the business itself. It means that, although some overhead can be inserted by reconfiguration mechanism, the service composition adaptation is better than its execution producing undesirable behaviors.

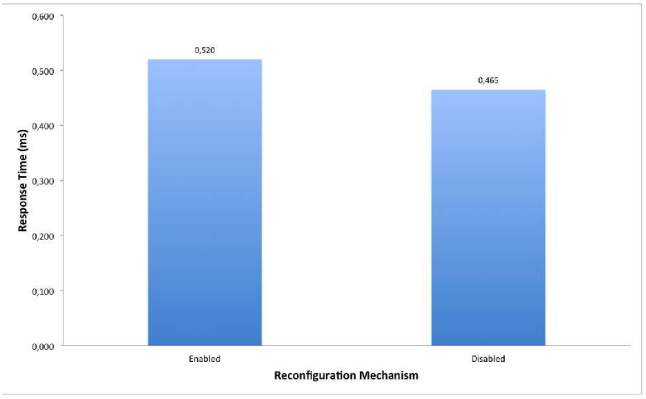

Figure 6. Overhead caused by the reconfiguration mechanism

The reconfiguration time expresses the time spent from the detection of an undesirable behavior until the end of the service composition adaptation. The goal is analyzing if there are divergent variations regarding the time spent in carrying out the different reconfigurations. The collected data prove that the average time to perform the reconfigurations is 55ms, and the pattern deviation is 3,1ms. In total, undesirable behaviors were injected into the solution, which permitted that 166 reconfigurations were carried out. Some of these configurations showed a longer reconfiguration time, such as the reconfigurations 56 to 61. For these cases, reinvocations and service replacements were subsequently performed. Regarding the cases with lower values, the undesirable behavior problem was solved by just reinvoking the service, which characterizes it as transient. Figure 7 depicts this time considering the 166 consecutive reconfigurations, which shows that the reconfiguration time is stable. It is important to highlight that most of this time is used because the CADP tool needs to check the suitable property.

VIII. Related Works

We introduce related work in two groups. First, those approaches specifically related to the monitoring and verification of service composition. Second, approaches that are focused on some aspects of adaptive service composition.

A. Monitoring and verification

Most studies about checking are focused on behavior verification using formal description techniques. For example, [13] propose a method called WSCMon to monitor web service composition. In WSCMon, WS-BPEL processes are automatically transformed into Communicating Sequential Processes (CSP) processes as system specifications and

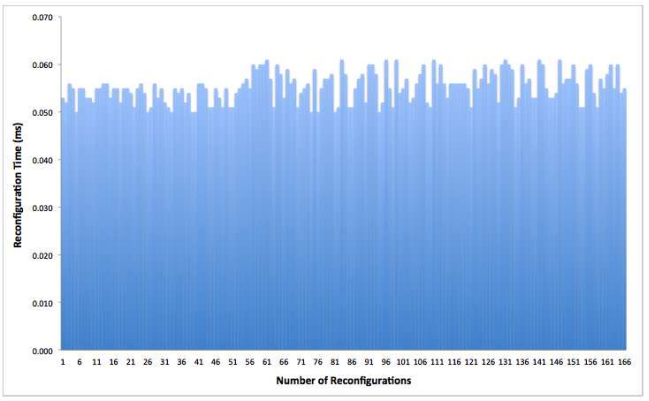

Figure 7. Reconfiguration time

behavioral properties are specified using CSP or LTL. Although these works deal with verification of functional requirements at runtime through formal techniques, none of them are focused on adaptation. Besides, some of them are bonded to specific notations such as the WS-BPEL.

[14] present a framework that supports the runtime verification of behavioral properties or assumptions about the behavior. Properties are automatically extracted from WS-BPEL specification in event calculus (EC) and are represented by EC formulas. Execution trace has the same format; then, it is possible to verify the conformity between trace and behavioral properties. Other studies work in a similar way; for example, [15] transform a WS-BPEL specification into a Colored Petri Net (CPN) to capture concurrent behavior and a runtime monitor using the CPN description to detect inappropriate use of protocol. [16] adopted stream X-machine (SXM) formalism to model the dynamic behavior. A testing method applicable to SXMs is capable of deriving test sets, which can prove the correctness of the implementation. The focus of these works is also the service composition behavior verification at runtime, which is different from the proposal presented in this work, where the focus is on adaptation.

Besides verifying the behavior, some studies focus on non-functional requirements. For instance, [17] propose a framework to support the monitoring of functionality and quality of service requirements which are specified as part of service level agreements. Specifications of service guarantee the terms that are specified in EC-Assertion, which is based on Event Calculus (EC). Other researches also detect runtime faults for web service composition but without employing formal description techniques. For example, [18] present the application of runtime monitoring to detect problems based on fault handling mechanism. Differently from these works, our proposals focus is the use of formal methods to verify service composition behavior at runtime.

B. Adaptation

Many proposals cope with dynamic reconfiguration, but they focus on different aspects of adaptation. For instance, some approaches are concerned with providing adaptation to service composition described in WS-BPEL. For example, [19] focus on exploring some WS-BPEL features which can be used for dynamic reconfiguration of service composition (e.g. scopes). The work [20] proposes Dynamo, an approach that allows the addition of adaptation capabilities for BPEL processes through AOP-techniques (Aspect-oriented). [21] also propose a plug-in architecture for self-adaptive web service composition, where self-adaptation features are well-modularized in aspect based plug-ins. The focus of our work is not the service composition adaptation by using WS-BPEL. Instead, we propose that our solution can be used for compositions described in any language, as long as they generate the necessary execution trace.

Besides, some researches provide support for dynamic reconfiguration based on distinct non-functional aspects as KAMI framework [22], which focuses on non-functional properties quantitatively specified in a probabilistic way by Markov models, such as reliability and performance. Petri nets also have been used to help choose the best configuration with the optimal quality of services (QoS) to meet users non-functional requirements [23]. Other techniques have also been used. For instance, [24] elaborated a well-founded model and theory of adaptation introducing written formalisms using COWS for evaluating properties such as reliability and responsiveness. Differently from the previous proposals, our solution focus is not on adaptation based on non-functional-requirements.

Furthermore, some proposals are concerned with identifying reconfiguration plans for adaptive service composition. For instance, [25] introduced a methodology and a tool for learning the services repairment strategies and selecting repairment actions automatically. For this purpose, it used a Bayesian classification of faults. Our work is concerned about dealing with the entire adaptation process, and not only with using specific reconfiguration plans.

Nonetheless, despite the number of works published about adaptive service composition, most of them do not deal with (or are not concerned about) the analysis of services based on execution traces and in the lightweight use of formal tools.

IX. Conclusions

This paper presented an approach for building selfadaptive service composition that has the following features: it uses formal methods in the lightweight way, runtime monitoring and verification of the functional requirements and reactive adaptation. Our contributions in this paper include the definition of an approach where existing service composition systems are analyzed by means of execution trace and the lightweight use of formal techniques for modelling the desirable behavior of service composition. The approach was presented by using a running example and its experimental evaluation demonstrated for assessing the impact that the system adaptation has upon the systems′ performance.

Future works will focus on the validation of the functional and non-functional properties, such as performance and availability. Also, certain behaviors must be predicted and the necessary adaptations must be carried out before the behavior occurs.

References

[1] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-oriented computing: State of the art and research challenges," *IEEE Computer*, vol. 40, no. 11, pp. 38–45, 2007.

[2] C. Zeginis and D. Plexousakis, "Web service adaptation: State of the art and research challenges," Institute of Computer Science, FORTH-ICS, Heraklion, Crete, Greece, Tech. Rep. Technical Report 410, 2010.

[3] R. Babaee and S. M. Babamir, "Runtime verification of service-oriented systems: a well-rounded survey," *International Journal of Web and Grid Services*, vol. 9, no. 3, pp. 213–267, 2013.

[4] S. Easterbrook, R. Lutz, R. Covington, J. Kelly, Y. Ampo, and D. Hamilton, "Experiences using lightweight formal methods for requirements modeling," *IEEE Transactions on Software Engineering*, vol. 24, no. 1, pp. 4–14, 1998.

[5] J. Hielscher, A. Metzger, and R. Kazhamiakin, "Taxonomy of adaptation principles and mechanisms," Tech. Rep., 2009.

[6] N. Rosa, "Middleware reconfiguration relying on formal methods," in *Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing (CIT/IUCC/DASC/PICOM), 2015 IEEE International Conference on*, October 2015, pp. 648–655.

[7] J. Yang and M. P. Papazoglou, "Service components for managing the life-cycle of service compositions," *Information Systems*, vol. 29, no. 2, pp. 97–125, april 2004.

[8] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Property specification patterns for finite-state verification," in *Proceedings of the Second Workshop on Formal Methods in Software Practice*. ACM Press, 1998, pp. 7–15.

[9] R. Mateescu and M. Sighireanu, "Efficient on-the-fly model-checking for regular alternation-free mu-calculus," *Science of Computer Programming*, vol. 46, no. 3, pp. 255 – 281, 2003, special issue on Formal Methods for Industrial Critical Systems.

[10] L. Baresi, C. Ghezzi, and S. Guinea, "Towards self-healing composition of services," in *Contributions to Ubiquitous Computing*, ser. Studies in Computational Intelligence, J. B. Krmer and A. W. Halang, Eds. Springer Berlin Heidelberg, 2007, vol. 42, pp. 27–46.

[11] H. Garavel, F. Lang, R. Mateescu, and W. Serwe, "CADP 2011: a toolbox for the construction and analysis of distributed processes," *STTT*, vol. 15, no. 2, pp. 89–107, 2013.

[12] H. Verbeek, J. Buijs, B. van Dongen, and W. van der Aalst, "Prom 6: The process mining toolkit," in *Proceedings of BPM Demonstration Track 2010*, ser. CEUR Workshop Proceedings, M. L. Rosa, Ed., vol. 615, 2010, pp. 34–39.

[13] M. Khaxar and S. Jalili, "Wscmon: runtime monitoring of web service orchestration based on refinement checking," *Service Oriented Computing and Applications*, vol. 6, no. 1, pp. 33–49, 2012.

[14] K. Mahbub and G. Spanoudakis, "Run-time monitoring of requirements for systems composed of web-services: Initial implementation and evaluation experience." in *ICWS*. IEEE Computer Society, 2005, pp. 257–265.

[15] J. Zhu and F. Kordon, "A petri net based runtime monitoring method for web services specified with bpel," in *2nd International Conference on Information Management and Engineering (ICIME 2010)*. IEEE, april 2010, pp. 304–310.

[16] T. D. Cao, T. T. Phan-Quang, P. Felix, and R. Castanet, "Automated runtime verification for web services," in *Web Services (ICWS), 2010 IEEE International Conference on*, July 2010, pp. 76–82.

[17] K. Mahbub and G. Spanoudakis, *Test and Analysis of Web Services*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, ch. Monitoring WS-Agreements: An Event Calculus–Based Approach, pp. 265–306.

[18] K. Jayashree and S. Anand, "Run time fault detection system for web service composition and execution." *Smart CR*, vol. 5, no. 3, pp. 469–482, 2015.

[19] M. Mazzara, N. Dragoni, and M. Zhou, "Implementing workflow reconfiguration in ws-bpel," *Journal of Internet Services and Information Security (JISIS)*, vol. 2, no. 1/2, pp. 73–92, 2 2012.

[20] L. Baresi, S. Guinea, and L. Pasquale, "Self-healing bpel processes with dynamo and the jboss rule engine," in *International Workshop on Engineering of Software Services for Pervasive Environments: In Conjunction with the 6th ESEC/FSE Joint Meeting*, ser. ESSPE '07. New York, NY, USA: ACM, 2007, pp. 11–20.

[21] A. Charfi, T. Dinkelaker, and M. Mezini, "A plug-in architecture for self-adaptive web service compositions," in *ICWS*. IEEE Computer Society, 2009, pp. 35–42.

[22] A. Filieri, C. Ghezzi, and G. Tamburrelli, "A formal approach to adaptive software: Continuous assurance of non-functional requirements," *Formal Aspects Computing*, vol. 24, no. 2, pp. 163–186, 2012.

[23] P. Xiong, Y. Fan, and M. Zhou, "A petri net-based approach to qos-aware configuration for web services." in *SMC*. IEEE, 2007, pp. 1286–1291.

[24] J. Fox, "A formal model for dynamically adaptable services," *CoRR*, vol. abs/1011.2652, 2010.

[25] B. Pernici and A. M. Rosati, "Automatic learning of repair strategies for web services." in *ECOWS*. IEEE Computer Society, 2007, pp. 119–128.