A Flexible Architecture to Monitor Dynamic Web Services Composition

Flavio Corradini, Francesco De Angelis, Daniele Fanı' and Andrea Polini
Computer Science Division, University of Camerino, MC, Camerino, Italy

Keywords: System Integration, Service Oriented Architecture and Methodology, Web Services and Web Engineering, SOA, Run-Time Monitoring, WS Choreography.

Abstract: A Service Oriented Architecture aims to facilitate interaction of loosely coupled services in large-scale dynamic systems. Despite a decade's active research and development, Web Services still remain undependable (Bourne et al., 2012). In literature many proposals attempt to overcome interoperability issues, particularly typical of not-orchestrated Web Service (WS) compositions. Although these techniques aid to discover potential interoperability mismatches, they do not fit well with flexibility and dynamism, desirable characteristics e.g. in choreographies. Here unsafe run-time changes may compromise a correct execution. To support such dynamism and to mitigate the effects of such failures, we propose a flexible architecture able to realize dynamic WS compositions, supporting run-time monitoring and verification techniques. The technique we chose is a novel run-time algorithm capable to predict potential failures that can happen in near future states of a choreography. It admits an integration "a-priori" and monitors the run-time services behaviour to provide information about possible errors when these can happen.

1 INTRODUCTION

Nowadays software is not anymore a single artifact that can be fully designed at design-time. Building complex systems from scratch is a very costly effort and often not successful. The current trend is to assemble a system similarly to a jigsaw puzzle. Once requirements and policies are established, existing software components are discovered and assembled to reach the prefixed goal. But whereas jigsaw puzzle pieces are made pluggable into each other by the same company, the systems to be integrated rarely are. Utilizing existing components means expecting that they will safely interact with each other, even if made independently by different companies. When properties as flexibility and dynamism are required, interoperability issues become even more complicated. For instance this is the case of SOA-based and Future Internet-oriented applications (Baresi et al., 2006). This kind of systems are made mostly by already existing WSs that provide just their public interfaces over internet. They may come from partially unknown third parties, thus further limiting the trust on their correctness and unmalicious behavior.

Such complex systems made by composing WSs as they were components, can be grouped in two main categories: orchestrated and not-orchestrated. In both of them, WSs are viewed as black-box components that interact with each other to reach the system goal. Briefly, an orchestration of WSs has one central coordinator (orchestrator). It manages the message exchange among the services following an executable business process. On the other hand, a not-orchestrated service composition like a choreography, focuses on the global protocol running between WSs rather than on their local coupling. Each component here knows requirements to interact with the others, and covers just at run-time a pre-fixed role. Assuming that a set of composable WSs is available at run-time in a system, they should interact on their own to reach the choreography goal. It is easy to see the additional effort required to grant robustness and safe interactions in a system without a central point of coordination. However, the lack of centralized coordination permits a more flexible system, where components can be dynamically bounded at run-time to a choreography role. This means that a WS could potentially enter, leave or be substituted while the whole system is being operated.

In this setting two main problems limit the possibility of software-quality related activities to operate in advance. One is the possible late integration of services in dynamic compositions, that makes it difficult to check the correctness of the resulting composition before the execution. On the other hand, the number of all the possible configurations for such

Corradini F., De Angelis F., Fani' D. and Polini A..
A Flexible Architecture to Monitor Dynamic Web Services Composition.
DOI: 10.5220/0005444800640072
In *Proceedings of the 11th International Conference on Web Information Systems and Technologies* (WEBIST-2015), pages 64-72
ISBN: 978-989-758-106-9

systems may be unmanageable according to (Morin et al., 2009), and it may become impossible to validate all of them in advance. Moreover not all the services expose their internal behavior, so projections or empirical behaviors are assumed for the composition validation. This can lead to unsafe run-time changes that may imply interoperability issues, since new interactions may emerge. In such a context, traditional testing and static analysis strategies may not be enough. In literature many proposals attempt to overcome interoperability issues. Many of them focus on the correct-by-construction strategy like (Autili et al., 2015), or on static verification like choreography realizability (Basu et al., 2012) and protocol compatibility (Elia et al., 2014). Although these techniques aid to discover potential interoperability mismatch, they do not fit well with characteristics like flexibility and dynamism of WS compositions. Moreover, often they are too time-expensive, especially with the emergence of short living compositions. In this scenario new verification techniques are emerging, that take advantage of both design- and run-time verification activities. Nevertheless, the underlying architecture must support them, in addition to the high dynamism of the system.

According to Cassandra (De Angelis et al., 2014), we think that it could be more fruitful to permit the services integration "a-priori", and then provide dynamic run-time validation and verification activities. Cassandra proposes a novel run-time monitoring and verification algorithm in order to detect possible failures in near future states of a choreography. The strategy identifies the current execution state, and then drives the construction of predictions that look to a number k of steps ahead of the current execution state. For this purpose a flexible architecture is needed, able to manage frequent run-time changes and that allows a verification strategy to monitor the system knowing the current execution state.

The goal of this research paper is to propose an architecture, hidden from the services point of view, able to manage dynamic WS compositions as choreographies, and to support run-time verification techniques like the above-mentioned Cassandra approach. This paper contributes to the state-of-the-art in online failure prediction of service composition in this respect: i) it introduces an architecture able to support integration-failures prediction at run-time in dynamic SOA systems; ii) it explains why existing services have not to be modified with our approach, externalizing message handling and monitoring; iii) it provides a concrete implementation of the approach, using Cassandra as assessment.

Section 2 provides some introductory material on Interface Automata, used to explain WSs composition. Then the proposed architecture is explained in 3. Section 4 introduces the Cassandra approach, since it has been used as verification technique in our concrete implementation described in Section 5. Related work are discussed in Section 6 while Section 7 concludes the paper and provides a list of future research directions.

2 THEORETICAL BACKGROUND

This section provides a theoretical background to explain what is a WS composition and problems that may occur. To ensure that two WSs can interact in a composition, as defined later in this section, their internal behavior must be *composable*. We can consider a WS behavior as an *Interface Automaton (IA)*, introduced in (de Alfaro and Henzinger, 2001), i.e. a light-weight formalism for modeling temporal aspects of software components interfaces. *IA* interact through the synchronization of input and output actions, and asynchronously interleave all the other (i.e. internal) actions. Below we provide some basic *IA* definitions.

Definition 1. *An* interface automaton *is a tuple* $P = \langle V_P, V_P^{init}, \mathcal{A}_P^I, \mathcal{A}_P^O, \mathcal{A}_P^H, \mathcal{T}_P \rangle$ *where:*

- V_P *is a set of* states *and* V_P^{init} *is a set of* initial states *that contains at most one state.*
- $\mathcal{A}_P^I, \mathcal{A}_P^O$ *and* $\mathcal{A}_P^H$ *are mutually disjoint sets of* input, output *and* internal actions. *We define* $\mathcal{A}_P = \mathcal{A}_P^I \cup \mathcal{A}_P^O \cup \mathcal{A}_P^H$.
- $\mathcal{T}_P \subseteq V_P \times \mathcal{A}_P \times V_P$ *is a set of steps.*

We say that an action $a \in \mathcal{A}_P$ is enabled at a state $v \in V_P$ if there is a step $(v,a,v') \in \mathcal{T}_P$. $\mathcal{A}_P^I(v)$, $\mathcal{A}_P^O(v)$ and $\mathcal{A}_P^H(v)$ are the subsets of input, output and internal actions that are enabled at v and $\mathcal{A}_P(v) = \mathcal{A}_P^I(v) \cup \mathcal{A}_P^O(v) \cup \mathcal{A}_P^H(v)$ is the set of action enabled at v. A key feature is that *IA* are *not* required to be input-enabled, i.e. we do not assume that $\mathcal{A}_P^I(v) = \mathcal{A}_P^I$ for each $v \in V_P$. The inputs in $\mathcal{A}_P^I \setminus \mathcal{A}_P^I(v)$ are called illegal inputs at v. The set of *input steps* is the subset of steps $\mathcal{T}_P^I = \{(v,a,u) \in \mathcal{T}_P \,|\, a \in \mathcal{A}_P^I\} \subseteq \mathcal{T}_P$. Similarly we define the sets $\mathcal{T}_P^O$ and $\mathcal{T}_P^H$ of *output* and *internal steps*. Moreover, a state u is reachable from v if there is an *execution sequence*, namely an alternating sequence of states and actions of the form $v = v_0, a_0, v_1, a_1, \ldots, v_n = u$ where each (v_i, a_i, v_{i+1}) is a step.

Two *IA* are *mutually composable* if their set of actions are disjoint, except that some input actions of one automaton can be output actions of the other one.

Definition 2. *Two interface automata P and Q are* mutually composable *(composable, for short) if $\mathcal{A}_P^H \cap \mathcal{A}_Q = \mathcal{A}_Q^H \cap \mathcal{A}_P = \emptyset$ and $\mathcal{A}_P^I \cap \mathcal{A}_Q^I = \mathcal{A}_P^O \cap \mathcal{A}_Q^O = \emptyset$. Essentially, P and Q are composable whenever they only share some input and output actions and, hence,* $\mathsf{shared}(P,Q) = \mathcal{A}_P \cap \mathcal{A}_Q = (\mathcal{A}_P^I \cap \mathcal{A}_Q^O) \cup (\mathcal{A}_Q^I \cap \mathcal{A}_P^O)$.

If two *IA* P and Q are composable, their product $P \otimes Q$ is a *IA* which set of states is $V_P \times V_Q$ and that will synchronize on shared actions, while asynchronously interleave all other (i.e. internal) actions. Since P and Q are not required to be input-enabled, their product $P \otimes Q$ may have one or more states where one component produces an output that the other one is not able to accept. The states where this happens are called *illegal*.

Definition 3. *All pairs $(v,u) \in V_P \times V_Q$ where there is an action $a \in \mathsf{shared}(P,Q)$ such that either $a \in \mathcal{A}_P^O(v) \backslash \mathcal{A}_Q^I(u)$ or $a \in \mathcal{A}_Q^O(u) \backslash \mathcal{A}_P^I(v)$, are called* illegal.

Illegal states represent error states that the composed system should not be able to reach. Indeed, two interface automata P and Q can be used together if there is at least a legal environment, i.e. an environment that can prevent $P \otimes Q$ from entering its illegal states.

We assume that each WS in a choreography provides a run-time model definition *IA*-equivalent, describing its behavior and how it interacts with the environment. In some real context this assumption can be judged a bit strong, but in literature it is easy to find techniques to derive such models. One of them, suggested by (Shoham et al., 2008), is a client-side mining of specifications based on static code analysis. Another technique could be the use of the Test Suites proposed by (De Angelis et al., 2013). They are used to verify if a service is able to cover a specific role in a choreography. If that is the case, we can consider the projection of that role as the run-time model of that service. However the real behavior could be more complex than the provided model, so unpredictable interactions could emerge, visible only monitoring at run-time. The architecture we propose in the next section supports both real-time monitoring and failure prediction techniques based on provided run-time models.

3 A FLEXIBLE ARCHITECTURE

This section introduces the structure of the architecture we propose. Its aim is to allow safer services integration "a-priori" in choreographies, since supported monitoring and predictions techniques will discover potential errors in time, without modifying the internal business logic of the participating services. The architecture adopts the Enterprise Service Bus (ESB) design pattern to make heterogeneous services exchange messages, while an integration framework is used to implement the choreography specification.

We assume that each participant service can be explicitly added and removed from the ESB registry. As explained later in this chapter, once added, each participating service will be bound to an ESB bundle according to the role it covers. Such a bundle will address messages and notify the monitoring algorithms, externalizing also error handling from the real services. We refer to the ESB as a single logic unit. However it could be distributed among several implementation to support scalability (Hanumantharayappa, 2014).

For our concrete architecture implementation, we chose "ServiceMix" (SMX) as ESB provider and "Camel" as integration framework. After a brief introduction to the just mentioned technologies, we describe the proposed architecture analyzing our concrete implementation.

3.1 Technological Background

3.1.1 Apache ServiceMix

Apache ServiceMix (http://servicemix.apache.org) is a flexible, open-source integration container. It provides a complete Enterprise Service Bus (ESB) powered by OSGi. An ESB is a software architecture model used to allow the communication between mutually interacting software applications in a service-oriented architecture (SOA). As opposed to the client-server model, the ESB promotes agility and flexibility. SMX can be run as a standalone provider or as a service within another ESB. It provides messaging, routing and Enterprise Integration Patterns (EIP) with Apache Camel, introduced in the next subsection. SMX is built on top of Karaf (http://karaf.apache.org), so it can be used also as OSGi container. We use this feature to install OSGi bundles to extend its features and routing rules.

3.1.2 Apache Camel

Apache Camel (http://camel.apache.org) is an open-source Java framework that allows easier integrations, implementing the standard EIPs using Java, XML, or Scala. It aims to address the problem of dealing with the specifics of applications and transports, and the problem to use a good solution to integration problems. Camel uses its *components*, that provide an Endpoint interface, to add connectivity to

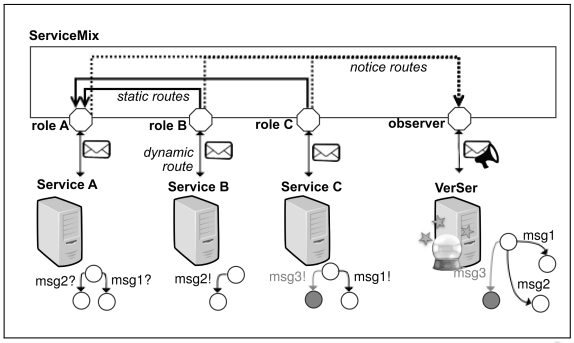

Figure 1: Implemented architecture.

different systems and different transports (http, rest, soap,...). To manipulate and mediate messages in between endpoints, Camel uses EIPs defined as *processors*. To wire *processors* and endpoints together, multiple DSLs languages can be used, such as Java, Scala or Spring.

3.2 The Architecture

The solution we propose in this paper, aims to build a flexible architecture able to support the execution of dynamic choreographies with real-time monitoring to discover unpredictable messages. Furthermore, it supports failure-predictions techniques based on provided WSs run-time model.

3.2.1 Realize a Choreography

We use SMX as the core component. It will be responsible to address the messages, connect the services and notify the verification algorithms. In our solution SMX is a standalone provider, accessible from all the potential participants of a choreography. Due to its OSGi extensible structure, together with Camel routing capability, it can realize a choreography specification. A choreography, regardless the language used to model it, specifies the roles and the communication rules. We represent roles in SMX with OSGi bundles we call *role-bundle*. So given a choreography, for each role a relative *role-bundle* is created and installed on SMX. The installation of bundles can be performed even at run-time. For now we can consider the *role-bundles* just as reference to roles, since each of them has a defined endpoint. They do not have a business logic, that instead is implemented by the service. Actually we will see later that the bundles can be used to perform actions such as transforming messages and sending notifications.

When a service enters in a choreography, we assume it provides its behavior (used by the prediction activities) and its endpoint to the ESB, in our case SMX. A route then is created at run-time between that service and the *role-bundle* relative to the role it covers. We call these routes *dynamic*, since they can change during a choreography execution. A *dynamic* route connects the *role-bundle* to the up-to-date endpoint of the current service performing that role. However, even if several different services can cover the same role over time, the endpoint of the relative *role-bundle* remains unchanged. The Fig. 1 shows the described architecture, representing the OSGi *role-bundles* as small octagons plugged into SMX.

All the messages of a service will be sent to the endpoint of the *role-bundle* relative to the role the service covers through *dynamic* routes. Services are not aware of the architecture neither of the *static* routing rules nor the endpoints of the other services. *Static* routes are those that implement the choreography specification, and they do not change along the choreography execution. They connect the *role-bundles* to each other, and address the messages following the choreography rules. When a *role-bundle* will receive a message from another *role-bundle* through a *static* route, it will send the message to the receiver service through a *dynamic* route.

The distinction between *static* and *dynamic* routes is just logic, they are defined in the same way and they are hidden from the outside. In our case, using Camel, they can be written in several languages including DSL-Java and Spring-XML. This allows to easily derive routes from a choreography, and easily update *dynamic* routes at run-time, according to current available services. In addition to routing, Camel allows also to handle messages before redirecting them to the receiver. Messages can be parsed and transformed, making interoperable even services implemented in different ways.

3.2.2 Verification Techniques Integration

We saw how a choreography could be realized with a very flexible and dynamic architecture. Now we are going to see how to use verification techniques like real-time monitoring and failure prediction. In our scenario we consider to have verification algorithms implemented in a remote service, we call *VerSer*, linked to an OSGi bundle of SMX, just like the other participant services. We call *observer* the bundle linked to *VerSer*, as shown by Fig. 1. To improve performance and to reduce latency, *VerSer* algorithms could also be implemented as an OSGi bundle, becoming part of SMX. We prefer to keep *VerSer* detached from SMX to be more flexible, and to be ready for distributed versions.

In order to make predictions, *VerSer* must be aware of the services behavior and of the messages

actually exchanged by each other. For this purpose, the *role-bundles* must notify the *observer* about the messages received from the participating services. The *observer* in turn notifies *VerSer* making it able to monitor the message exchange. Besides the exchanged messages, also actions like ”add” or ”remove” of a service must be notified to *VerSer*.

In our proposed architecture, each participant must provide its *Interface Automaton* equivalent run-time model. When the service enters in the system, its run-time model will be sent to *VerSer* using a special message. Likewise, when a service leaves the choreography, a special message is sent to *VerSer* in order to update its prediction mechanisms. Notifications are automatically sent by *role-bundles* to the *observer* through camel routes we call *notice-route*, just as the other routing rules. The *notice-routes* are represented in Fig. 1 by dashed arrows. This solution permits participating services to ignore the verification mechanisms. If *VerSer* predicts a potential future failure, it will notify the concerned *role-bundle*. This permits to externalize error-handling techniques, implementing them in the *role-bundles* without modify the existing services.

Referring to the example of Fig. 1, the use of the routes to send the ”msg1” from the service C could be the following. When the WS C wants to send the ”msg1” to A, the message is sent to the fixed endpoint of *role-bundle* C. Then, the message is forwarded to the observer through the *notice* route RoleC→Observer. At this point the Observer redirects ”msg1” to VerSer through the *dynamic* route Observer→VerSer, so that the latter can monitor the system and update its prediction. Then the prediction is sent back to the fixed endpoint of the Observer, that redirects it to the concerned *role-bundle* C. If no failure will be predicted, i.e. no protocol mismatches will be expected performing ”msg1”, the message will be sent to RoleA through the *static* route RoleC→RoleA. In turn, the message will be forwarded through the *dynamic* route RoleA→ServiceA.

To assess our architecture we use cassandra, described in Sec. 4, as verification technique to be deployed as *VerSer*.

4 CASSANDRA

To test our architecture with real monitoring and prediction activities, we chose CASSANDRA (De Angelis et al., 2014), an approach to forecast possible failures in the dynamic integration of software components. The usual approach to the modeling of a component that interacts with other components is *pessimistic*; the basic assumption is that the environment can behave as it likes and that two components can be composed if no environment can lead them into an error state. Cassandra approach is optimistic, permitting services integration ”a-priori”, and then providing run-time monitoring and verification activities. It explores design-time system models together with events observed at run-time to check if a possible illegal state is reachable. The exploration strategy builds a global design-time model (*exploration tree*) looking just k steps ahead starting from the current system state monitored at run-time. In this paper we assume that the value of k is defined by the user. In order to make near future prediction, Cassandra should have an up-to-date view over the participating services and the exchanged messages. For example, if a participant leaves the choreography at run-time, the notification of the ”remove” will allow Cassandra to update the illegal states of the *exploration tree*.

4.1 Cassandra Overview

This online failure prediction algorithm relies on a specification of the component behaviour based on the *IA* formalism shown in Sec. 2. The algorithm suitably composes the specifications of those components under execution. Nevertheless, instead of using the classical composition operator, Cassandra uses a slightly different one more suitable for the online prediction of failures in a dynamic environment. According to this composition operator any pair of components sharing a set of I/O actions can be integrated. Then, the composed automata is navigated by looking ahead to the current execution state. It is the task of the failure prediction approach to check whether the system is approaching an *illegal state*, and so to inform a possible failure avoidance mechanism that will possibly take care of repair actions. In the Cassandra approach, an illegal state corresponds to an integration failure by any path shorter than k steps and originating in the current state. In a sense, we assume that an illegal state can be reached as consequence of a wrong invocation/message done by one component on a component that either is not willing to accept it in the current state or does not exist at all.

Fig. 2 shows a client-server scenario used to explain the Cassandra algorithm. The *sender* asks the *server* to write or read on *receiver*’s file. The services cannot always correctly cooperate since they make different assumptions on their respective behaviour. The *sender* assumes that after a successful opening, the resource can be used through the server without receiving any failure. The *server* may connect with the *receiver* through a not reliable medium, and can

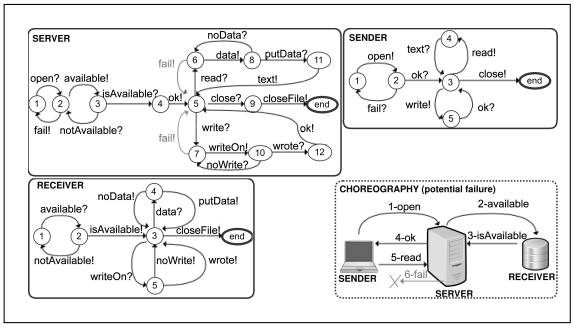

Figure 2: Scenario client-server.

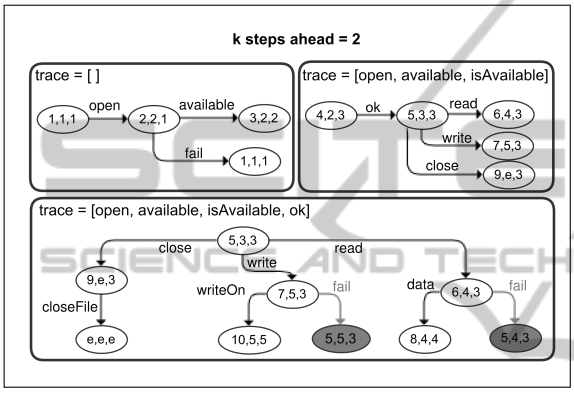

Figure 3: Cassandra interaction tree.

return a failure. According to the standard *IA* theory the two automata cannot be composed, since the *server* may send a $fail$ message while the *sender* is not able to receive it. However Cassandra let them interact, since if no connection failures occur, the *sender* will successful access the *receiver*'s file. A snippet of the *exploration tree* derived by Cassandra is shown by Fig. 3, assuming a prediction of 2 steps ahead. The color of illegal states is red.

4.2 Cassandra as VerSer

Cassandra can be integrated in our architecture as the *VerSer*, creating a *dynamic* route to the *observer*. As Cassandra does, we assume the services behavior *IA*-equivalent, are described by a SCXML (State Machine Notation for Control Abstraction) specification, i.e. a general-purpose event-based state machine language. When a service participate in a choreography, it is added to the SMX registry and a *dynamic* route is created towards the relative *role-bundle*. Moreover, a special message is sent to Cassandra, notifying the "add" of the service and providing its SCXML specification. This process is showed by Figure 4, where the service "C" referenced from Fig. 1 is going to participate in a choreography, providing its endpoint

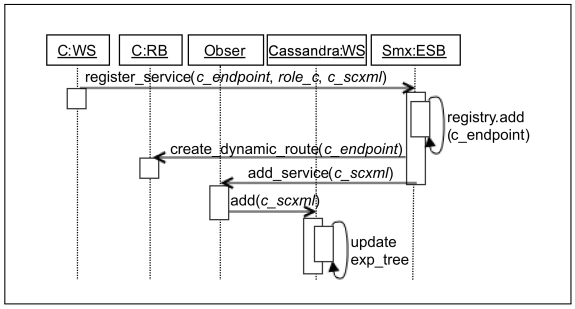

Figure 4: Service registration.

(*c_endpoint*), the role it wants to cover (*role_c*) and its behavior (*c_scxml*). In the same way, when a service is explicitly removed from the SMX registry, a special message is sent to Cassandra which updates its *exploration tree* with new illegal states. Services can be added and removed even at runt-time, during a choreography execution. The characterization of the states in the *exploration tree* will result from the observed events and the actual configuration of the system to make predictions used by a fault management strategy or used for logging and analysis purpose without altering the services implementation. During the execution of the system, at logical level each participant asks Cassandra for a prediction *before* sending a message to the receiver service. The prediction is returned to the participant synchronously to the system execution and the management of the possible futures is left to the service. Actually this strategy is implemented by routing rules and *role-bundles* behaviors. Indeed, each *role-bundle* will redirect messages from services to the *observer* through the *notice* routes asking for a prediction. Once the prediction is available, an answer is sent back by the *observer*. The *role-bundle* then, if no illegal states are reached by the *exploration tree*, it will deliver the message to the receiver role through the *static* route, or it will handle the potential failure otherwise.

This strategy allows to check in advance if an action can be performed safely, i.e. it does not lead to a failure. It checks that the receiver service is active and its current state is able to accept that message from the sender. This strategy increases latency of the system, but grants system consistency and safer interactions. As stated in (De Angelis et al., 2014), the complexity of Cassandra failure prediction algorithm mainly depends on the size (in terms of number of nodes and edges) of the *exploration tree*. Its creation and its exploration have a time complexity in $O(N^k)$, so it increases according the number of steps to forecast in the future (which relates to the value of k), rather than the number of total interactions in the system.

5 ASSESSMENT

This section analyzes a scenario we used as use-case for assessment of Cassandra with the architecture we proposed. For the sake of readability we used a very simple scenario, with the intent to show the feasibility of the architecture. The scenario models a Single-Sign-On choreography, with three roles: user agent (*Usr*), Identity Provider (*IdP*) and Service Provider (*SP*). The *Usr* requests the access to a resource to the *SP*, that in turn asks to the *Usr* an authentication token. Then the *Usr* asks the *IdP* to be authenticated. If the user is successful authenticated, the token will be sent back in the response. Only at this point the *Usr* can access to the *SP* resources. This scenario is illustrated by Fig. 5.

We assume that Cassandra is aware of the services behavior, represented by SCXML files. Though our solution is independent from the services implementation, we chose to implement services as Java remote Akka actors. Akka (http://akka.io) is an open-source actor-based toolkit and runtime supporting Java and Scala. It simplifies the construction of concurrent and distributed applications. An *actor* in Akka is the smallest unit of an application. It is a container for State, Behavior and Mailbox. A State, similar to an explicit state machine, is used to compare the Cassandra prediction with the services real current state. The Behavior is the action to be taken, according to the current state and the message received. The Behavior is defined with an event-based approach, making actors interact through asynchronous messages, enqueued in their Mailboxes. Akka provides also self-healing techniques if an *actor* reaches an inconsistent state, recovering a previous state. We chose Akka because of the features listed above, allowing us to consider them as remote services with an exposed behavior. Even the Cassandra algorithm has been implemented as an Akka actor, installed directly in ServiceMix as an OSGi bundle without the *observer*. In our demo we have three remote Akka actors and ServiceMix that provides three *role-bundles* and one bundle acting as a local Cassandra service. Actually all the components are deployed in the same machine (2,4 GHz Intel Core 2 Duo, 4GB ram) but with different endpoints to simulate a distributed system. Different tests have been performed on the SSO scenario, changing the prediction steps ahead (k). Analyzing costs of the tests, we got the values listed by Tab. 1. Analysis are made on executions of 9 seconds.

With the scenario we used, our tests proved the efficacy of the proposed architecture with the failure prediction approach Cassandra. Though, more complex systems should be tested in real evolving distributed scenarios. We are confident in positive results even with complex scenarios since, as stated in Sec. 4, the time complexity of the predictions is not strictly related to the number of interactions of a system.

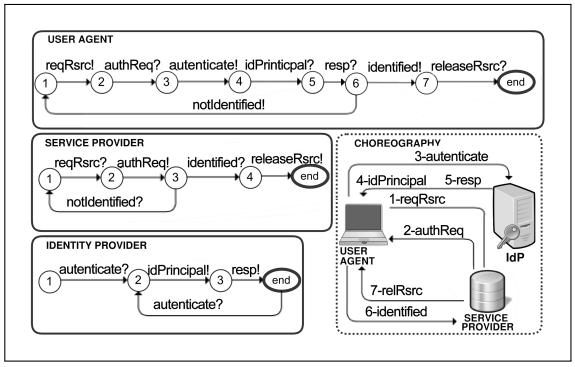

Figure 5: Scenario SSO.

Table 1: Resources consumption over time of async monitoring. Scenario of Fig. 5.

k	4s		7s		9s	
	cpu	ram	cpu	ram	cpu	ram
k=0	1%	20MB	1%	20MB	1%	20MB
k=2	6%	30MB	5%	30MB	5%	30MB
k=5	7%	30MB	5%	50MB	5%	50MB

6 RELATED WORK

This section provides related work on failure prediction, and run-time verification of dynamically evolving systems.

The most significant paper on online failure prediction can be most probably considered the survey conducted by Salfner et al. in (Salfner et al., 2010). This survey analyzes and compares a number of existing online failure prediction methods, proposing a taxonomy to classify the existing online failure prediction methods. Most of them use heuristics, statistics or probabilistic models to predict potential future failures. We developed Cassandra as online failure prediction for choreographies, but the presented architecture could also be used in general dynamically evolving systems.

Our proposed solution could be adapted to be used even with different approaches from Cassandra. A considerable number of approaches have been developed for run-time monitoring of dynamically evolving systems, as in (Chatley et al., 2004; Barringer et al., 2007). The authors in (Chatley et al., 2004) show how it is possible to generate a snapshot of the structure of a running application, and how this can be combined with behavioral specifications for com-

ponents to check compatibility against system properties. In (Barringer et al., 2007) it is described a mechanisms for combining programs from separate components and an operational semantics for programmed evolvable systems. In (Goldsby et al., 2007) a run-time monitoring and verification technique is proposed, which can check whether dynamically adaptive software satisfies its requirements. In (Filieri et al., 2012) KAMI is proposed to support fault detection. It shares similar goals with Cassandra, even if we focus on functional failures and them on non functional properties. The work in (Ghezzi et al., 2012) shares with Cassandra monitor functional properties in dynamically evolving systems for inferring a run-time model of it. without failure prediction.

7 CONCLUSIONS AND FUTURE WORK

This paper has proposed a concrete architecture to realize dynamic WS compositions, capable of supporting run-time failure prediction with the Cassandra approach. This approach captures the current state of a service-based system through monitoring its execution to check potential protocol mismatches in the near future. The proposed architecture, based on an ESB and an Integration Framework, allows services to be dynamically composed, exchanging messages following choreography rules. The whole system can be monitored at run-time, without modifying the participating services. Our demo, although tested on a light scenario, demonstrates a concrete implementation to make use of the Cassandra algorithm. We are confident that the approach is quite promising, even if technical and theoretical problems must be still solved.

As future work, we want to investigate a technique to automatically derive *role-bundle* and *static* routes from a choreography specification. Moreover, we want to use our architecture with monitoring capabilities to automatic repair at run-time unrealizable choreographies. As we allow the a-priori integration of services, we want to optimistically realize choreographies, even if they may be not-realizable. Our idea is to provide role-bundles with automatic choreography-repair techniques, activated if possible interoperability errors are predicted.

ACKNOWLEDGEMENTS

This work has been partially supported by:

- "Open City Platform" - funded by MIUR (SCN 00467);
- "CHOReOS" - EU FP7 Project (IP 257178).

REFERENCES

Autili, M., Inverardi, P., and Tivoli, M. (2015). Automated synthesis of service choreographies. *Software, IEEE*, 32(1):50–57.

Baresi, L., Nitto, E. D., and Ghezzi, C. (2006). Towards open-world software: Issue and challenges. In *SEW-30 2006, 25-28 April 2006, Columbia, MD, USA*, pages 249–252.

Barringer, H., Gabbay, D. M., and Rydeheard, D. E. (2007). From runtime verification to evolvable systems. In *RV*, pages 97–110.

Basu, S., Bultan, T., and Ouederni, M. (2012). Deciding choreography realizability. In *ACM SIGPLAN Notices*, volume 47, pages 191–202. ACM.

Bourne, S., Szabo, C., and Sheng, Q. Z. (2012). Ensuring well-formed conversations between control and operational behaviors of web services. *Service-Oriented Computing*, pages 507–515.

Chatley, R., Eisenbach, S., Kramer, J., Magee, J., and Uchitel, S. (2004). Predictable dynamic plugin systems. In *FASE*, pages 129–143.

de Alfaro, L. and Henzinger, T. A. (2001). Interface automata. In *ESEC/SIGSOFT FSE*, pages 109–120.

De Angelis, F., Di Berardini, M. R., Muccini, H., and Polini, A. (2014). Cassandra: An online failure prediction strategy for dynamically evolving systems. In *Formal Methods and Software Engineering*, pages 107–122. Springer.

De Angelis, F., Fanì, D., and Polini, A. (2013). Partes: A test generation strategy for choreography participants. In *Automation of Software Test (AST), 2013 8th International Workshop on*, pages 26–32. IEEE.

Elia, I. A., Laranjeiro, N., and Vieira, M. (2014). Itws: An extensible tool for interoperability testing of web services. In *Web Services (ICWS), 2014 IEEE International Conference on*, pages 409–416. IEEE.

Filieri, A., Ghezzi, C., and Tamburrelli, G. (2012). A formal approach to adaptive software: continuous assurance of non-functional requirements. *Formal Aspects of Computing*, 24:163–186.

Ghezzi, C., Mocci, A., and Sangiorgio, M. (2012). Runtime monitoring of component changes with spy@runtime. In *ICSE 2012*, pages 1403 –1406.

Goldsby, H., Cheng, B. H. C., and Zhang, J. (2007). Amoeba-rt: Run-time verification of adaptive software. In *MoDELS Workshops*, pages 212–224.

Hanumantharayappa, A. K. (2014). Enabling horizontal scalability in an open source enterprise services bus.

Morin, B., Barais, O., Jézéquel, J.-M., Fleurey, F., and Solberg, A. (2009). Models@ run.time to support dynamic adaptation. *IEEE Computer*, 42(10):44–51.

Salfner, F., Lenk, M., and Malek, M. (2010). A survey of online failure prediction methods. *ACM Comput. Surv.*, 42(3).

Shoham, S., Yahav, E., Fink, S. J., and Pistoia, M. (2008). Static specification mining using automata-based abstractions. *IEEE Trans. Software Eng.*, 34(5):651–666.