SCENE: A Service Composition Execution Environment Supporting Dynamic Changes Disciplined Through Rules

Massimiliano Colombo[1], Elisabetta Di Nitto[1,2], and Marco Mauri[1]

[1] CEFRIEL, Via Fucini 2, 20133 Milano – Italy
[2] Politecnico di Milano, Piazza Leonardo da Vinci, 32, 20133 Milano - Italy
mcolombo@cefriel.it, dinitto@elet.polimi.it, mmauri@cefriel.it

Abstract. Service compositions are created by exploiting existing component services that are, in general, out of the control of the composition developer. The challenge nowadays is to make such compositions able to dynamically reconfigure themselves in order to address the cases when the component services do not behave as expected and when the execution context changes. We argue that the problem has to be tackled at two levels: on the one side, the runtime platform should be flexible enough to support the selection of alternative services, the negotiation of their service level agreements, and the partial replanning of a composition. On the other side, the language used to develop the composition should support the designer in defining the constraints and conditions that regulate selection, negotiation, and replanning actions at runtime. In this paper we present the SCENE platform that partially addresses the above issues by offering a language for composition design that extends the standard BPEL language with rules used to guide the execution of binding and re-binding self-reconfiguration operations.

Keywords: service composition, autonomic behavior, self-reconfiguring systems, dynamic binding.

1 Introduction

Service-oriented approaches are capturing a growing interest not only as a mean for business to business integration, but also as the possible reference architecture to support the development of systems exposing autonomic and dynamically changing behavior [11]. Typical examples are the cases of applications able to reconfigure themselves and to contact different services depending on contextual information (e.g., the location of the final user), on QoS levels, on possible failures happening while a service is running, and so on.

The dynamic nature of such systems precludes the a-priori identification of the services defining the system and demands for run-time *discovery* and *selection* of such services. In particular, we argue that discovery and selection have to be supported at two different levels. On one side, the runtime platform executing systems built by composing services should be flexible enough to support the discovery and selection of alternative services, and the negotiation of their service level agreements. On the other side, the language used to develop the composition should support the

A. Dan and W. Lamersdorf (Eds.): ICSOC 2006, LNCS 4294, pp. 191 – 202, 2006.

designer not only in the definition of the way service invocations are sequenced in a workflow, but also in the definition of self-configuration policies that will discipline the selection of services and negotiation actions at runtime.

Through these policies, it should be possible, for instance, to express the fact that whenever a component service breaks the Service Level Agreement (SLA) [6] it has established with the system, an attempt to establish a new SLA is done and, if this fails, then the system will try to find an alternative component service, preferably offered by the same provider.

Industrial composition environments, typically BPEL-based [3], offer little support to dynamic changes [4] and do not support the explicit definition of self-configuration policies. If we look at service-oriented research initiatives, they tend to encapsulate self-configuration policies in some infrastructural components rather than to allow the designer to define them explicitly. In these cases, the degree of flexibility of the resulting system is often poor and designers have little control on the policies that are actually applied at runtime (see Section 6 for more details on some of these approaches).

In this scenario, we propose a composition language through which we describe service compositions in terms of two distinct parts: a process part, described using BPEL, that defines the main business logic of the composition, and a declarative part, described using ECA (Event Condition Action) rules. Rules are used to associate a BPEL workflow with the declaration of the policy to be used during (re)configuration. Rules can either be defined at design time or later before the execution of the system. Moreover, various sets of rules can coexist and be activated depending on the preferences of the system users.

A composition written in our language is executed by SCENE (*Service Composition ExecutioN Environment*). The current implementation of SCENE integrates an off-the-shelf BPEL executor called PXE [15] and a rule engine called Drools [9]. SCENE is part of a European project called SeCSE (*Service Centric Systems Engineering*) [17] aiming at providing methods, tools, and platform to support service-oriented engineering.

For the moment both language and runtime environment support the dynamic discovery and selection of services. We are currently working to provide support to the other self-configuration actions we have mentioned (e.g., negotiation of new SLAs and replanning of the composition).

The goal of this paper is to present the results we have achieved so far. Consistently, the paper is organized as follows: in Section 2 an example is shown to motivate our work. Section 3 briefly describes the main aspects of the composition language. Section 4 proposes the architecture of SCENE together with its execution semantics. Section 5 presents a preliminary evaluation of the approach, Section 6 presents the related approaches, finally, Section 7 concludes this paper and discusses about some possible future work.

2 Motivating Example

In this section we present an example that motivates the need for supporting dynamic changes in a composition. The example has been extracted from the scenarios defined by the industrial partners of the SeCSE project.

A car maker would like to equip its top level cars with a device allowing the end user to exploit a set of remote services all available through a portal offered by the car maker itself. One of the offered services, named *XTRIP*, helps the end user in keeping his/her schedule updated depending on the status of his/her travel. In particular, the service allows the user to plan a trip. Based on the plan and on a navigation system that allows the service to know the geographical position of the car, the service itself is able to automatically check the agenda of the user to make sure that he/she will be on time for the scheduled appointments. In case of problems, the service automatically establishes a telephone communication between the end user in the car and his/her secretary so that they can take actions to change the schedule as needed.

The process realizing the *XTRIP* service is composed of activities (see the activity diagram of Fig. 1) to get the data about the current position of the car (*plan trip*), to access the user's agenda (*check for conflicts* and *confirm commitments*) and to establish a telephone call when needed (*make call*).

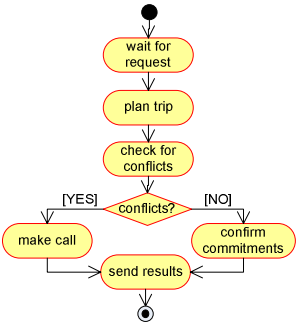

Fig. 1. The XTRIP process

The completion of these activities implies the invocation of operations to external services capable of fulfilling them. *check for conflicts* and *confirm commitments* activities are provided by the same service as they both operate on the end user agenda. The first one checks if some appointments for the period of interest are conflicting with the trip, and the second one confirms the appointments in case no conflict is detected.

Despite the simplicity of the example, its implementation should be dynamically adaptable to the way the service is actually used. In particular, the selection of the actual service for *plan trip* activity could be left open at design time and decided at runtime based on the geographical location of the user. This way, it is possible to take advantage of navigation systems specialized for specific areas. Moreover, the selection of the concrete service to complete both *check for conflicts* and *confirm commitments* activities should depend on the user which requests the service. Therefore, the service cannot be fixed at design time. Finally, the selection of the service to satisfy *make call* activity could depend on the telecom provider that offers the best rate to connect the traveler with his/her secretary and also on performance; these, again, are issues that cannot be addressed while designing the composition workflow.

By exploiting our composition approach, at design time all these choices can be left open, but, at the same time, rules can be defined that will guide all (re)configuration actions that will be taken at runtime.

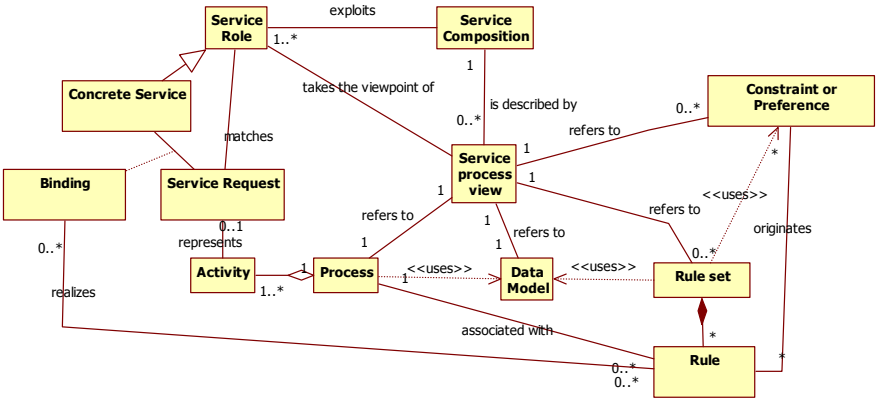

Fig. 2. The constituents of a service composition

3 The SCENE Composition Language

The constituents of a service composition in the SCENE composition language are shown in Fig. 2. A *Service Composition* can exploit various *Service Roles* and can be described by various *Service Process Views* each one describing the composition from the view point of a specific role in the composition.

The service process view is composed of four main elements: a *Process* that in our language is described using BPEL, one or more *Rule Sets* and *Constraints* and *Preferences* to control the self-reconfiguration of the composition at runtime, and a *Data Model* that includes all data types that are shared between the process and the rules. These include the structure of the input and output data used by the process part to communicate with the external services.

The process is composed of A*ctivities* that can be implemented as *Requests* to external services. These requests can be served by some service roles. At runtime, the rules are used to realize the *Bindings* between service requests and the concrete implementation of service roles (*Concrete Services*).

The decoupling between the process and the rules allows for a proper separation of concerns. The process is described in BPEL. We do not pose any restriction to the structure of the BPEL code describing the process. It defines the data and control flow among the various elements of the composition. The rules and the constraints and preferences define the policies used for self-organization. They, in fact, contain all it is needed to dynamically select/change services, possibly discovered on the fly.

Rules are aware of the state of the process. On the contrary, the process is completely agnostic of the existence of the rule part. As we will discuss in Section 4, at runtime, the execution environment is in charge of communicating to the rule part the state of the process and of modifying this state (and in the future work also the structure of the process) depending on the results provided by the execution of rules.

Rules do not have to be necessarily defined together with the process; they can also be introduced at any time before execution in order to account for the specific conditions in which the process is being executed. The notation we propose for rules is an extension of the XML rule language that is interpreted by Drools (see next section).

```
<!-- processInfo and eventList are global variables -->
<rule name="makeCall - prefix different from 33, 34, 38">
        <scope>
                <activity-name>make call</activity-name>
        </scope>
        <event>
                <name>bindingEvent</name>
                <type>rulelanguage.datamodel.events.ActivityBindingEvent</type>
        </event>
        <condition>
            !(bindingEvent.getInputVariableValue("CalledParty").equals("33") ||
              bindingEvent.getInputVariableValue("CalledParty").equals("34")  ||
              bindingEvent.getInputVariableValue("CalledParty").equals("38"))
        </condition>
        <consequence description="dynamic discovery and binding action request created and sent as an event payload">
                        DynamicDiscoveryBindingActionRequest actionRequest = new
                        DynamicDiscoveryBindingActionRequest(processInfo.getLocalBindingPreferences("makeCall"));
                        actionRequest.setActivityName("make call");
                        eventList.addExternalEvent(new Event(actionRequest));
        </consequence>
</rule>
```

Fig. 3. An example of dynamic discovery and binding rule

Fig. 3 shows an example of rule that refers to the case study described in Section 2. The rule can be activated while the composition process is executing the *make call* activity. The aim of the activity is to establish a phone call between the user in the car and his/her assistant. It exploits a Third Party Call Control (TPCC) service [20] offered by some telecom provider. The aim of the rule shown in Fig. 3 is to allow the activity to be dynamically bound to the most suitable TPCC service. More in detail, the rule is triggered by an event of type *ActivityBindingEvent* issued by the process execution environment whenever it realizes that the service request associated with some process activity is not bounded to any specific service. The condition that activates the rule requires that the phone number to be called through the TPCC service (this is the parameter of *bindingEvent*) does not start with some special numbers. These, in fact, are managed by a different rule (not shown for space reasons). In the action part a request for executing a dynamic discovery and binding operation is created (*actionRequest*) and this request is wrapped into an event that, in turn, is added to an event queue to be actually issued. The event queue is identified by the variable *eventList* that is defined as global and visible to all rules. Variable *processInfo* is global as well. It contains the defined self-reconfiguration preferences and constraints.

In general, rules can introduce some changes in the composition or they can delegate specific tasks to some infrastructural components. The rule sample presented above falls in this last category since it aims at activating the component able to identify new bindings (the Binder as explained in Section 4). This component receives as input some binding preferences through the variable *processInfo*. Fig. 4 shows a fragment of binding preferences stating that the service to be selected has to guarantee a price per minute of telephone conversation lower than 0.5 euros. Of course, in order to guarantee this preference, the selection of the concrete service to be bound has to be limited to all those exposing pricing information in their service specification.

The binding preferences fragment also provides information about the validity of each binding. In the example, the binding is valid for a single invocation. This means that it has to be renewed each time the corresponding process activity is executed. A determined binding can also be valid for the lifetime of the specific process instance or it can be associated with all new instances of the same process. In these last two cases, the *bindingValidity* tag takes the values *PROCESS INSTANCE* and *PERMANENT* respectively.

```
...
<local-binding-preferences>
        <discoveryPreferences> ... </discoveryPreferences>
        <selectionPreferences>
                <preference>
                        <key>singleQualityConstraint</key>
                        <value>PRICEPERMIN &lt; 0.5 (€) </value>
                </preference>
        </selectionPreferences>
        <bindingValidity>SINGLE INVOCATION</bindingValidity>
</local-binding-preferences>
...
```

Fig. 4. Binding preferences

Fig. 5 shows a sample of rule that triggers the reconfiguration of the composition in response to an event generated by a monitoring component. This reconfiguration can happen independently of the validity that has been set for bindings since it is used to deal with faulty situations: the event triggering the rule signals that some failure in a component service has occurred. The service that has failed is stored in a "blacklist" for future record and a *CacheUpdatingActionRequest* is issued (again, it is wrapped in a proper event) that causes any current binding to the failing service to be actually eliminated.

Rules can have either global or local scope. Rules having global scope can be activated any time during the execution of a composition. The service violation rule of

Fig. 5 is an example of these kinds of rule. Rules with local scope are associated with some specific activities of the process that require interaction with external services. This means that these rules can be activated and can affect the composition only when the corresponding activity is being executed. The dynamic binding rule of Figure 3 has local scope (defined in the rule *scope* tag) since it is built to deal with a specific binding to a service offering a *makeCall* operation.

Fig. 6 shows some constraints defined on the *confirm commitments* activity. For each invoke activity the constraints specify if the activity is abstract, i.e., it has not been bound to a concrete service operation, if it is concrete, i.e., a binding for it has been finalized at design-time, and if any rule associated with that invoke activity is enabled. A relevant aspect concerns the presence of potential dependencies between activities.

```
<!-- processInfo and eventList are global variables -->
<rule name="service violation event">
        <scope>
                <all/>
        </scope>
        <event>
                <name>violationEvent</name>
                <type>rulelanguage.datamodel.events.ServiceViolationEvent</type>
        </event>
        <consequence description="the service endpoint is stored into the endpoint black list">
                processInfo.updateBlackList(violationEvent.getServiceDetails());
                CacheUpdatingActionRequest actionRequest = new CacheUpdatingActionRequest();
                actionRequest.setServiceEndpoint(violationEvent.getServiceDetails().getEndpoint());
                actionRequest.setMode("delete");
                eventList.addExternalEvent(new Event(actionRequest));
        </consequence>
</rule>
```

Fig. 5. Violation rule

In the specific example of Fig. 6, the dependency tag indicates that the activity under consideration (*confirm commitments*) has to rely on the same service exploited by the *check for conflicts* activity. Other kinds of dependencies can be expressed in terms of generic constraints (e.g., the service used in the current activity has to offer the same level of performance of the service used in some other activity).

```
<activity-info>
        <activity-name>confirm commitments</activity-name>
        <realization-info>ABSTRACT</realization-info>
        <binding-dependencies>
                SAME SERVICE AS 'check for conflicts'
        </binding-dependencies>
        <rules-enabled>true</rules-enabled>
        <!--a rule will be auto-generated from this dependency-->
</activity-info>
```

Fig. 6. Dependencies between activities

4 Architecture of SCENE and Execution Model

The SCENE platform provides the runtime execution environment for compositions written in the SCENE language. The first prototype includes the following components (see Fig. 7):

- REDS [5], a publish-subscribe infrastructure that acts as integration middleware and supports both synchronous and asynchronous multicast communication.
- A process execution environment that, in turn, is composed of an open source BPEL engine, PXE, which is in charge of executing the process part of the service composition [15] and of a set of Proxies that decouple the BPEL engine from the logic needed to support reconfiguration (see below).
- An open source rule engine, Drools, responsible for running the ECA rules [9].
- The Binder, responsible for executing binding actions at runtime based on the directions defined in the rule language. This component is able to execute various policies for selecting the candidate services. For instance, the services could be selected from a predefined list or they could be selected in the "outside world" by exploiting some discovery mechanism. The selection could be based both on functional and non-functional attributes. More details on the Binder component can be found in [8].
- A monitoring system [1] is also connected to the bus and provides SCENE with the needed monitoring feedbacks.

Other components are being added to the platform to manage aspects such as dynamic negotiation and dynamic reconfiguration of a service composition.

In the following we briefly show how the SCENE platform supports the execution of a composition. We start from a situation where the designer has defined the composition process by using the standard BPEL constructs. He can have either defined all bindings to some concrete services or he can have left these undefined into the BPEL process. Also the designer might have defined some constraints to enable/disable the association of process activities with rules, to define dependencies between activities. When applicable, he defines rules and binding preferences that will be considered during the self-reconfiguration of the composition.

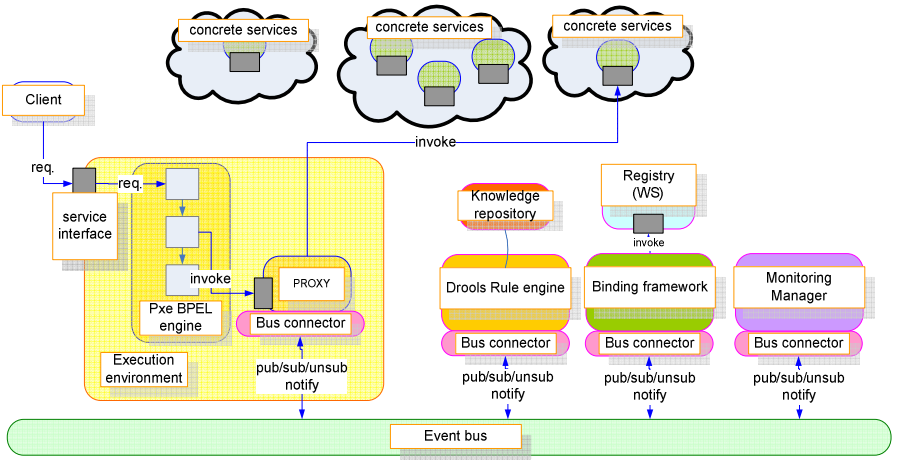

Fig. 7. The architecture of the SCENE platform

At deployment time, a SCENE composition is preprocessed. All activities in the BPEL process having the rule-enabled tag set to true are bound to specifically instantiated proxies. Moreover, additional rules are automatically generated that enforce the constraints defined as part of *processInfo*. For instance, the rule in Fig. 8 is generated to guarantee that the dependency relation defined between *confirm commitments* and *check for conflicts* activities is actually respected.

At runtime, when the execution of the process reaches the invocation of an external service, an operation offered by the proxy bound to that activity is actually called. The proxy has a well defined internal logic that works as follows: if the proxy refers to a valid concrete service, then it propagates the request directly to this service. In the opposite case, it emits on the bus an *ActivityBindingEvent*. The rule engine -- that has subscribed to this event -- receives it and activates a rule able to handle the missing binding (if this rule exists). If more than one rule can be activated, only one is non-deterministically selected (for the future we plan to add priorities to let the designer have an influence on this selection). The activated rule can take any decision, ranging from the activation of the binding procedure (as done by the rule in Fig. 3) to the immediate binding to some previously identified service (as done by the rule in Fig. 8), to the termination of the execution.

In all cases, when a new service is ready to be bound to the composition, or a decision to stop the composition is taken, an event is generated by the rule engine and received by the proxy that can then change its internal state, possibly invoke the proper operation on the bound service (if it has been identified), and then pass the control back to the BPEL execution environment. In the case a permanent binding has been identified, the rule engine will also load the files containing the process and change them by adding a concrete binding to the selected service.

When the monitoring infrastructure detects misbehavior of some service, it issues an event of type *ServiceViolationEvent*. Again, this is received by the rule engine and

```
<!-- processInfo and eventList are global variables -->
<rule name="autogenerated rule">
        <scope>
            <activity-name>confirm commitments</activity-name>
        </scope>
        <event>
            <name>bindingEvent</name>
            <type>rulelanguage.datamodel.events.ActivityBindingEvent</type>
        </event>
        <consequence description="">
            String endpoint = processInfo.getBinding("check for conflicts");
            ActivityBindingResultActionRequestactionRequest =
                        new ActivityBindingResultActionRequest();
            BindingResultInfo bindingInfo = new BindingResultInfo();
            bindingInfo.setAccessPoint(endpoint);
            actionRequest.setBindingInfo(bindingInfo);
            processinfo.updateBindingInfo_forActivity("confirm commitments", endpoint);
            eventList.addExternalEvent(new Event(actionRequest));
        </consequence>
</rule>
```

Fig. 8. Rule automatically generated to account for dependency between *confirm commitments* and *check for conflicts* activities

activates a violation rule. As in the previous case, various decisions can be defined in the rule to recover from the faulty situation. These range from simply disregarding the failure to change one or more bindings in the composition. In the forthcoming version of SCENE, we will manage also dynamic negotiation of SLAs and structural transformation of the composition itself.

5 Evaluation

We have developed some case studies to test the flexibility of the approach and its applicability to concrete cases. The examples we have considered so far have been defined within the SeCSE consortium and concern automotive and telecom domains. The example presented in Section 2 is one of these. For that example, besides the BPEL process sketched in Fig. 1, we have defined 15 rules. Some of the rules are those we have presented in the previous sections. The others have a similar structure and are not reported here for the sake of space.

The example, exploits some real services that are actually offered in a pre-operational environment by our partners. For instance, the TPCC service is offered by Telecom Italia and actually exploits the communication machinery of the company.

For the example we have developed two GUIs not shown here for space reasons. One of them is owned by the consumer of the XTRIP service and the other by the administrator. The consumer can start the execution of the composed service by requesting the system to plan a trip. Together with the request, the GUI has to pass to the composition some data about the user (his/her current position, the trip destination, the agenda service of the user, his phone number and the secretary's phone number). Through the administration interface can see the bindings and re-bindings that are computed during the execution of the composition can be monitored.

Our partners in the project are currently experimenting with the language and the platform and are providing feedbacks to us especially concerning the user friendliness of the language and its ability to capture their requirements. For the moment, this

analysis has not revealed major weaknesses in the ability of the language to express the partners' needs, and we are constantly working on improving the intuitiveness and simplicity of the language.

As a final remark we highlight that the execution of a SCENE composition introduces some overhead with respect to the execution of a plain BPEL process. More in detail, referring to a specific invocation, this overhead varies depending on the following cases:

- The designer has disabled the usage of rules for the current invocation activity: in this case, there is no overhead introduced by SCENE since the invocation is executed directly by the BPEL engine.
- The usage of rules is enabled and the invoke activity that is being executed has associated a valid binding (this is stored into the corresponding proxy). In this case, the proxy acts as an intermediary between the BPEL engine and the actual services. Being the proxy fully dedicated to a single service, the overhead is mainly concerning the message exchange between the engine and the proxy. These two are installed on the same machine.
- The usage of rules is enabled and the invoke activity that is being executed does not have associated a valid binding. In this case, the proxy receiving the invocation request triggers the binding procedure by generating an event for the rule engine. In this case the overhead cannot be determined a priori, but it depends on the complexity of the rules that are triggered. Reasonably, this complexity is compensated, however, by the ability of the system to reconfigure itself.
- The monitoring system (that can exist independently of SCENE to monitor the execution of standard BPEL processes) signals a fault. This, again, triggers the execution of rules and, again, introduces, an overhead that is compensated by the fact that the system may be able to return in a correct state.

6 Related Work

Various approaches in the service-oriented domain tend to add some kind of dynamic binding features to service composition. MAIS [7] supports dynamic binding but the logic for selecting candidate services is predefined and cannot depend on user inputs as it happens with our binding rules.

Meteor-S [19] supports the execution of binding operations at design time, deployment time, and just before the execution. It also takes into account binding dependencies [18]. The composition is divided into scopes; semantic web languages are used to describe both domain constraints and services; matchmaking algorithms are used to associate each scope with the concrete set of services to invoke. While this approach requires that a semantic description is attached to each potential component service, our approach can work both using a complete description for services or a more lightweight one. Moreover, we support runtime bindings that do not seem to be addressed in Meteor-S.

The approach presented in [12] exploits DAML-S technologies to support semantic discovery of services and their runtime integration into the composition. In this case, however, the approach does not account for dependences between service invocations.

SELF-SERV [2] exploits a proprietary language for describing a composition and introduces the concept of service community. Binding is possible at runtime among the members of the community.

In [13] composition rules are used to govern the way a composition is built in a semiautomatic way. Our approach differs from this because more than exploiting rules to build a new composition, we use them to support its runtime self-configuration.

Combining rules with workflow languages in a service-oriented context has been already proposed in the literature as a way to define conditional business logic that is not directly captured by the workflow [16, 10]. Our rules, indeed, are not designed to encapsulate some business logic. Instead, they work at a lower level of abstraction to support the definition of policies for dynamic binding (and negotiation and replanning in the future).

As we have tried to convey in the previous sections, our approach aims at offering some autonomic features. In the research area of autonomic computing, the main idea behind the scene is to build applications capable of self-managing themselves, reflecting the behavior of biological systems. An autonomic application must own the following properties [14]: self-Awareness, self-Configuring, self-Optimizing, self-Healing, self-Protecting, Context-Awareness, Openness, Anticipatory behavior. We think at least three of the aforementioned characteristics are satisfied by our approach: rules associated with the process makes the composition self-configuring (e.g., a binding rule can re-configure the association between a process activity and a concrete service), self-healing (e.g., a service violation event can trigger a recovery action rule to avoid the use of services whose measured QoS properties deviate from our requirements), context-aware (e.g., service compositions can be executed by means of several external services, on the basis of the knowledge of the user input and the actual available services, obtainable only at run-time). Considering the differences between our work and the application computing view, rules we use are not applied only to re-establish the equilibrium between environment and application, but also to delay the association of the activities to be executed with the concrete services till runtime.

7 Conclusion

In this paper we have focused on the definition of proper linguistic and infrastructural mechanisms to support self-configuration of a service composition.

As future work we plan to extend the language and the platform to support dynamic negotiation of service level agreements with component services and to drive dynamic changes in the structure of the composition itself.

We also need to continue with the evaluation of the approach for what concern both performances and usability.

Acknowledgements

This work is framed within *SeCSE* [17], IST Contract No. 511680. We thank all our partners in the project for their valuable comments.

References

1. L. Baresi and S. Guinea, “Towards Dynamic Monitoring of WS-BPEL Processes”, In the Proceedings of the 3rd International Conference of Service-oriented Computing (ICSOC'05). Amsterdam, The Netherlands, 2005.
2. B. Benatallah, M. Dumas, and Q. Z. Sheng, “Facilitating the Rapid Development and Scalable Orchestration of Composite Web Services”, *Distributed and Parallel Databases*, 17(1): pp. 5-37, Jan. 2005.
3. BPEL. “Business Process Execution Language for Web Services Version 1.1”, http://www.ibm.com/developerworks/library/ws-bpel/. May 2003.
4. S. Carey, “Part 3: Making BPEL Processes Dynamic”, SOA Best Practices: The BPEL Cookbook, OTN Oracle Web Site.
5. G. Cugola, and G. P. Picco, REDS: A Reconfigurable Dispatching System. Technical report, Politecnico di Milano, 2005.
6. A. Dan, et al., “Web Services on demand: WSLA-driven Automated Management”, *IBM Systems Journal*, Volume 43, Number 1, pages 136-158, IBM Corporation, March, 2004.
7. V. De Antonellis, M. Melchiori, L. De Santis, M. Mecella, E. Mussi, B. Pernici, P. Plebani, “A layered architecture for flexible e-service invocation”, *Software-Practice & Experience*. ISSN: 0038-0644, John Wiley & Sons, 2005.
8. M. Di Penta, R. Esposito, M. L. Villani, R. Codato, M. Colombo, and E. Di Nitto, “WS Binder: a Framework to enable Dynamic Binding of Composite Web Services”, in the Proceedings of the *ICSE Workshop on Service-Oriented Software Engineering* (IW-SOSE06), Shanghai China May 2006.
9. Drools. Java rule Engine. http://drools.org/.
10. K. Geminiuc, “Part 1: A Services-Oriented Approach to Business Rules Development”, SOA Best Practices: The BPEL Cookbook, OTN Oracle Web Site.
11. IBM, “Autonomic computing: Enabling Self Managing Solutions”, SOA and autonomic computing, IBM Whitepaper, Dec. 2005.
12. D. J. Mandell and S. A. McIlraith, “Adapting BPEL4WS for the Semantic Web: The Bottom-Up Approach to Web Service Interoperation”, in the Proceedings of the Second International Semantic Web Conference (ISWC2003), Sanibel Island, Florida, 2003.
13. B. Orriens, J. Yang, and M.P. Papazoglou, “A Framework for Business Rule Driven Service Composition”, in the Proceedings of the *3rd VLDB-TES Workshop*, Berlin, September 2003.
14. M. Parashar, and S. Hariri, “Autonomic Computing: An Overview", UPP 2004, Mont Saint-Michel, France, Editors: J.-P. Banâtre et al. LNCS, Springer Verlag, Vol. 3566.
15. PXE BPEL engine. http://www.fivesight.com/pxe.shtml.
16. F. Rosenberg, and S. Dustdar, “Towards a Distributed Service-Oriented Business Rules System”, in the Proceedings of *IEEE European Conference on Web services* (ECOWS), 14-16 November 2005, IEEE Computer Society Press.
17. SeCSE Website: http://secse.eng.it/.
18. K. Verma, R. Akkiraju, R. Goodwin, P. Doshi, J. Lee, “On Accommodating Inter Service Dependencies in Web Process Flow Composition”, in the Proceedings of *AAAI Spring Symposium on Semantic Web Services*, 2004.
19. K. Verma, K. Gomadam, A. P. Sheth, J. A. Miller, and Z. Wu, "The METEOR-S Approach for Configuring and Executing Dynamic Web Processes", Tech. Report 2005.
20. 3GPP, Technical Specification Group Core Network, Open Service Access (OSA), “Parlay X Web Services; Part 2: Third Party Call (Release 6)”, 3rd Generation Partnership Project Technical Specification 29.199-2, v2.0.0 (2004-09).