A Framework for Rule-Based Dynamic Adaptation*

Ivan Lanese[1], Antonio Bucchiarone[2], and Fabrizio Montesi[1]

[1] Lab. Focus, Università di Bologna/INRIA, Bologna, Italy
{lanese,fmontesi}@cs.unibo.it
[2] Fondazione Bruno Kessler - IRST, Trento, Italy
bucchiarone@fbk.eu

Abstract. We propose a new approach to dynamic adaptation, based on the combination of *adaptation hooks* provided by the adaptable application specifying where adaptation can happen, and *adaptation rules* external to the application, specifying when and how adaptation can be performed. We discuss different design choices that have to be considered when using such an approach, and then we propose a possible solution. We describe the solution in details, we apply it to a sample scenario and we implement it on top of the language Jolie.

1 Introduction

Adaptation, evolvability and reconfiguration are hot topics today. Adaptable systems change their behavior, reconfigure their structure and evolve over time reacting to changes in the operating conditions, so to always meet users' expectations [3]. This is fundamental since those systems live in distributed and mobile devices, such as mobile phones, PDAs, laptops, etc., thus their environment may change frequently. Also, user goals and needs may change dynamically, and systems should adapt accordingly, without intervention from technicians.

To achieve the required degree of flexibility, different research groups have proposed frameworks for programming more adaptable applications [1,13,20,17,23]. For instance, the application code may include constraints on the environment conditions or on the user behavior, and may specify how to change the application logic if those constraints are violated [5]. This approach is called *built-in adaptation*, and allows to adapt the application if the conditions change in some expected way. However, since the adaptation logic is hard-wired into the application, it is not possible to adapt to unforeseen changes in the operating conditions. *Dynamic adaptation* instead aims at adapting the system to unexpected changes [4]. Dynamic adaptation is challenging since information on the update to be performed is not known at application development time.

We propose a new approach to dynamic adaptation, based on the separation between the application and the adaptation specification. An adaptable

* Research supported by Projects FP7 EU FET ALLOW IST-324449, FET-GC II IST-2005-16004 Sensoria and FP7-231620 HATS.

M. Wirsing, M. Hofmann, and A. Rauschmayer (Eds.): TGC 2010, LNCS 6084, pp. 284–300, 2010.
© Springer-Verlag Berlin Heidelberg 2010

Table 1. List of possible (Travelling) domain activities

Activity	Functional Parameters			Non-functional Parameters	
Activity Name	Number	Source	Destination	Time	Cost
Take Train	IC2356	Bologna Train Station	Trento Train Station	2 h 41 m	20 euros
Take Bus	13	Trento Train Station	Univ. of Trento	30 m	1 euro
Take Taxi	25	Trento Train Station	Univ. of Trento	10 m	15 euros
Go To Meeting	-	Bob's House	Univ. of Trento	4 h	50 euros

application should provide some *adaptation hooks*, i.e. information on part of its structure and its behavior. The adaptation logic should be developed separately, for instance as a set of adaptation rules, by some adaptation engineer, and can be created/changed after the application has been deployed without affecting the running application. Adaptation should be enacted by an *adaptation manager*, possibly composed by different *adaptation servers*. At runtime, the adaptation manager should check the environment conditions and the user needs, control whether some adaptation rule has to be applied to the application, and exploit the adaptation hooks provided by the application to reconfigure it.

We describe now a scenario that will be used to validate our proposal.

1.1 Travelling Scenario

Consider Bob travelling from Bologna to University of Trento for a meeting. He may have on his mobile phone an application instructing him about what to do, taking care of the related tasks. A set of possible tasks are in Table 1. For instance, the activity *Take Train* connects to the information system of Bologna train station to buy the train ticket. It also instructs Bob to take the train.

Assume that such an application has been developed for adaptation. This means that its *adaptation interface* specifies that some of the activities are adaptable. Each adaptable activity has a few parameters, e.g. *Number* specifying the code of the train, bus or taxi to be taken, *Source* specifying the desired leaving place and *Destination* specifying the desired arrival place, all visible from the adaptation interface. Also, a few non-functional parameters for the activities may be specified as *Time* and *Cost*. We show now a few examples of adaptation.

Example 1. When Bob arrives to Bologna train station, its Travelling application connects to the adaptation server of the train station. Assume that a new "*FrecciaRossa*" (Italian high speed train) connection has been activated from Bologna to Trento providing a connection with *Time=1 h 23 m* and *Cost=32 euros*. This is reflected by the existence of an adaptation rule specifying that all the applications providing an activity *Take Train* for a train for which the new connection is possible may be adapted. Adaptation may depend on Bob's preferences for comparing the old implementation and the new one, or may be forced if, for instance, the old connection is no more available. In case adaptation has to be performed, the new code for the activity is sent by the adaptation

server to the application, and replaced as new definition of activity *Take Train*. Thus Bob can immediately exploit the new high speed connection, which was not expected when the application has been created.

Example 2. Suppose that the train from Bologna to Trento is one hour late. Bob mobile phone may have an adaptation server taking care of adapting all Bob's applications to changing environment conditions. The adaptation server will be notified about the train being late, and it may include an adaptation rule specifying that if Bob is late on his travel, he can take a taxi instead of arriving to the University by bus. The adaptation rule thus replaces the activity *Take Bus* of the travelling application with a new activity *Take Taxi*.

Example 3. Suppose that the train from Bologna to Trento is cancelled and there is no other train to reach Trento. Bob receives this event from the train station information system. The unique way to reach Trento is thus to rent a car (i.e., a *Rent a Car* activity) from his house. Bob's adaptation server provides adaptation rules specifying that if a resource needed by the activity *Go To Meeting* is not available, the whole activity has to be replaced by a different one. Again, adaptation depends on Bob's preferences for choosing the best offer from various car rental proposals. The code of the chosen one is sent by the adaptation manager to the application, and replaced for the *Go To Meeting* activity.

1.2 Overview

Both the described Travelling scenario and the technical solution that we propose can be applied to applications written in any language, e.g. Java, C++ or even BPEL. Thus we will discuss the general aspects in a language-independent setting, then move to more and more concrete settings to deal with aspects more related to the implementation. We will validate our approach using the Travelling scenario in Section 1.1 and the Jolie [16] language.

The main contributions of this paper are:

- the description of an approach able to realize the scenario;
- a discussion of different design choices for the realization of the approach;
- JoRBA (Jolie Rule-Based Adaptation framework), a proof-of-concept implementation based on the language Jolie.

The rest of the paper is structured as follows: Section 2 presents the rule-based approach that we propose to realize dynamic adaptation. In Section 3 we introduce the algorithm for enacting dynamic adaptations and we put it at work on the Travelling scenario. Section 4 describes JoRBA, a proof-of-concept implementation of our adaptation mechanisms based upon the Jolie language. The paper concludes with related works and conclusions.

2 A Rule-Based Approach to Dynamic Adaptation

In this section we discuss a possible solution for implementing the scenario described in Section 1.1. Our approach can be applied to applications developed

using any language, provided that (i) the application exposes the desired adaptation interface and (ii) the application is able to support the code mobility mechanism necessary for performing adaptation. In Section 4 we will show how this can be done for instance in the case of Jolie, a language for programming service-oriented applications.

Thus we want to build an *adaptable application* using some language L and following our approach to dynamic adaptation. The application must expose a set of *adaptable domain activities* (or, simply, activities) $\{A_i\}_{i\in I}$, together with some additional information. Activities A_i are the ones that may require to be updated to adapt the application to changes in the operating conditions. While it is necessary to guess where adaptation may be possible, it is not necessary to know at the application development time which actual conditions will trigger the adaptation, and which kind of adaptation should be performed.

The adaptable application will interact with an *adaptation manager*, possibly implemented as a set of *adaptation servers*, providing the adaptation rules. More precisely, it provides a set of rules $\{R_j\}_{j\in J}$, each of them specifying a possible adaptation. The environment has full control over the set of rules $\{R_j\}_{j\in J}$, and may change them at any time, regardless of the state of the running applications. Each such rule R includes a description D_R of the activity to be adapted, an applicability condition c_R specifying when the rule is applicable, the new code P_R of the activity, the set V_R of variables required by the activity, and some information NF_R on the non-functional properties of the activity.

At runtime, rule R is matched against application activity A to find out whether adaptation is possible/required. In particular:

- the description of the activity to be adapted in the rule (i.e., D_R) should be compatible with the description of the activity D_A in the application;
- the applicability condition c_R should evaluate to true; the applicability condition may refer to both variables of the adaptation manager and variables published by the adaptation interface of the application;
- the non-functional properties NF_R guaranteed by the new code provided by the adaptation rule should be better than the ones guaranteed by the old implementation, according to some user specified policy;
- the variables V_R required by the new code P_R should be a subset of the variables provided by the application for the activity.

If all these conditions are satisfied then adaptation can be performed, i.e. the new code of the activity should be sent by the adaptation manager to the application, and installed by the application replacing the old one. Since the update may also influence the state, we also allow the adaptation rule to specify a state update for the adaptable application.

Even if the approach presented so far is quite simple, a few technical decisions have to be taken to implement it in practice. We consider them below.

When is the applicability of rules checked?

We have two kinds of approaches to this problem: either adaptation is *application-triggered*, or it is *manager-triggered*.

If the adaptation is application-triggered then the application asks the adaptation manager to check whether activities have to be adapted. Application-triggered approaches ensure that updates are performed when they are more needed, and are best suited when the required updates strongly depend on the application state. There are three main possibilities:

On Initialization: when the application starts, it checks its environment for updates. This approach allows to build configurable applications, which adapt themselves to the environment where they are deployed. However, they do not react to changes in the environment occurring after start-up.

On Wait: when the application is idle, it checks for possible adaptations. This can be useful if the updates are particularly time-consuming.

On Activity Enter: when the application is about to enter a new activity, it checks whether some update for the activity to be executed is available. This is the most useful pattern, since activity enter is the last point in time when adaptation is possible. In this way one is guaranteed that the most updated implementation of the activity is executed. We do not deal with adaptation of ongoing activities, since this will require to consider how to adapt the state and the point of execution according to the current state of the activity. We leave the issue for future work.

Dually, adaptation can also be manager-triggered, i.e. the adaptation manager decides when to check whether applications need to be adapted. Manager-triggered approaches are best suited for adaptations that do not depend on the state of the application. Three approaches dual to the ones above are possible:

On Registration: when an application enters the range of an adaptation server, it registers onto the adaptation server itself, and rules are checked for applicability. This allows location adaptation, i.e. it allows to adapt mobile applications to the different environments they move in.

At Time Intervals: at fixed points in time, applicability of rules is checked for all the applications in the range. This allows to ensure that applications are updated within a predefined time bound.

On Rule Update: each time a new rule is added to the adaptation server, it is checked for applicability. This ensures that new rules are applied as soon as possible. This is useful for instance to update mobile applications before they leave the range of an adaptation server. The On Registration and On Rule Update approaches combined allow one to perform all the adaptations that are state-independent with the smallest possible number of checks.

The two kinds of approaches can be combined for maximal flexibility.

How to choose the order of application of rules?

Different rules may be applicable to the same activity at the same time. In this case the choice of which rule to apply first may change the time required for performing adaptation, or even the final result. For instance, if two updates enhance the same non-functional property, the best one will make the other

superfluous. In case of updates that do not influence the applicability of each other, only the last one will affect the final behavior.

Non-deterministic Update: updates are applied in non-deterministic order. This is the simplest possibility, in particular for distributed implementations. In fact, an application can be in the range of different adaptation servers, and this policy does not require them to synchronize (but for mutual exclusion during the updates). However this policy is applicable only in some cases, and may lead to troubles in others. Essentially, this approach is applicable when the order of application of the rules does not change the final result. A general analysis of when this approach is applicable is left for future work.

Priority Update: adaptation rules can be given a priority (static or dynamic): if many rules apply, the one with highest priority is applied first, and this forbids later applications of rules with lower priority. This approach guarantees that the smallest number of updates is performed, but it requires to check all the rules for applicability, and its distributed implementation is quite difficult. Some simplifications of the priority mechanism may be implemented in a more easy way. For instance, if one allows to apply rules in a random order, but does not want the effect of high priority rules to be reverted by low priority rules, it is enough to include priority of the adaptation rule as most significant factor in the non-functional properties, and to use the non-deterministic approach. Priority can also be applied only for rules managed by the same adaptation server, using the non-deterministic approach for rules of different servers. **Sequential Update**, where rules are ordered, is a particular case of priority update, where priority coincides with the position of the rule in the order.

Why is the application of an adaptation rule needed?

There are two classes of rules, and they have to be treated in different ways:

Corrective Rules: these rules take care of adapting the application when the current implementation is not viable in the current conditions; this is, e.g., the case of Example 3 in the Introduction, where the train has been cancelled. These rules will be identified thanks to a compulsory flag set to true, and they will be applied regardless of the non-functional properties of the previous available implementation.

Enhancing Rules: these rules enhance an existing activity which may however work; this may for instance change the non-functional properties of the activity, or provide new functionalities. These rules are identified since their compulsory flag is set to false, and they are applied only if the non-functional properties of the activity described by the rule are better than the ones of the previous activity according to the user-specified preferences. Even if the update does not involve (only) the non-functional behavior, the same approach can be used, since a version number can be added to the non-functional properties and used to distinguish the different levels of functionalities.

3 Algorithm and Example

In this section we formalize the algorithm for applying the adaptation rules, and discuss the example in more detail.

As we have seen, our adaptation framework is composed by two main interacting parts: a set of adaptable applications, each one exposing an adaptation interface, and an adaptation manager providing a set of adaptation rules.

Definition 1 (Adaptable application). *An* adaptable application *is an application exposing an adaptation interface (defined in more detail below) defining its set of domain activities and the public part of its state, and providing functionalities for accessing the public part of its state and for replacing its domain activities with external activities.*

Thus, the main feature of an adaptable application is its adaptation interface.

Definition 2 (Adaptation interface). *The* adaptation interface *of an application $\mathcal{A}$ is a set of quadruples $\langle D_A, V_A, NF_A, COMP_A \rangle$, one for each domain activity A, where D_A is a description of the activity, V_A its set of public variables, NF_A the values of non-functional properties provided by the current implementation, and $COMP_A$ a boolean function that given two sets of non-functional values decides which one is more desirable.*

The description D_A of the activity may have different forms. The only requirement is that it must be possible to check two descriptions for compatibility. We assume to this end a function $MATCH$, returning *true* if the two descriptions match, $false$ otherwise. For instance, one can consider as part of the description of each activity its *goal* in the sense of [22], and assume that two descriptions are compatible if the goal is the same. We will consider more complex definitions of activity compatibility in future work.

The set V_A in the adaptation interface contains the names of the public variables of the activity. They are used both to evaluate the applicability condition of the adaptation, which may depend on the current state of the application, and for the activity code to access the state of the whole application.

The set NF_A describes the non-functional properties of the activity A. It is a set of labelled values, one for each non-functional dimension. Examples of these dimensions can be ExecutionTime, Cost, SecurityLevel, and others. The set NF_A can be used also for more general purposes, including dimensions such as Priority and CodeVersion.

Function $COMP_A$ describes the user preferences concerning the non-functional properties of the activity. In particular, given two sets of non-functional properties, it checks which is the most desirable one. This function depends on the user preferences, thus it must be part of the starting application.

In the following we show examples of the adaptation interfaces of some activities introduced in Table 1.

Example 4 (Take Train Activity). The adaptation interface of the activity *Take Train* is a quadruple, defined as:

$$\langle D_{TakeTrain}, \{Number, Source, Destination\}, \\ \{Time = 161m, Cost = 20euros\}, COMP_{TakeTrain}\rangle$$

Here $D_{TakeTrain}$ is a description of an activity for going from *Source* to *Destination* using train number *Number.* These last are also the variables in the public interface of the activity. The actual values of those variables, which in this case are *Source=Bologna*, *Destination=Trento* and *Number=IC2356*, are retrieved at runtime from the state of the application. The third component describes the non-functional properties. In this case we have two dimensions, *Time*, describing the time required for the travel (in minutes) and *Cost*, describing the cost of the ticket (in euros). Finally, $COMP_{TakeTrain}$ is a function expressing the user preferences. For instance, given two pairs $\langle Time_1, Cost_1\rangle$ and $\langle Time_2, Cost_2\rangle$ it may return *true* (i.e., adaptation will improve) if $Time_2 < Time_1$ (i.e., the new solution is faster) and $(Cost_2 - Cost_1)/(Time_1 - Time_2) < 0,3$, i.e. each saved minute costs less than 30 cents.

Example 5 (Go To Meeting Activity). The adaptation interface of the activity *Go To Meeting* is a quadruple, defined as:

$$\langle D_{GoToMeeting}, \{Resources, Source, Destination\}, \\ \{Time = 240m, Cost = 50euros\}, COMP_{GoToMeeting}\rangle$$

Here $D_{GoToMeeting}$ is a description of an activity for going from *Source* (Bob's House) to *Destination* (University of Trento) using resources described by variable *Resources* (train IC2356, bus 13). These last are also the variables in the public interface of the activity. The time required is 4 hours while the cost is 50 euros. Finally, $COMP_{GoToMeeting}$ is a function expressing the user preferences.

The other component of our framework is the adaptation manager. It includes a state, which may be used to check the environment conditions (e.g., time, temperature, etc.), and the set of adaptation rules, one for each possible adaptation. It may be implemented in a distributed way as a set of adaptation servers.

Definition 3 (Adaptation Manager). *An* adaptation manager ρ *is a pair* $\langle V_\rho, \mathcal{R}_\rho\rangle$ *where* V_ρ *is a set of variables and* R_ρ *a set of rules.*

Each rule R *in* $\mathcal{R}_\rho$ *has the form* $D_R, c_R \vdash P_R(S_R, V_R, NF_R, CF_R)$ *where:*

- D_R *is a description of an activity,*
- c_R *is a boolean expression,*
- P_R *is a program,*
- S_R *is a state update,*
- V_R *is the set of variables required by* P_R *to work,*
- NF_R *is the set of non-functional properties guaranteed by* P_R*, and*
- CF_R *is the compulsory flag specifying whether the adaptation is compulsory.*

D_R describes the set of activities the rule can be applied to. This will be compared to the description D_A of the activity to be adapted. Condition c_R is the applicability condition for the rule: a boolean condition evaluated over the set of variables of the adaptation manager and the set of public variables of the activity. If this evaluates to *true* then the rule can be applied. P_R is the new code for the activity. P_R may use the public variables of the activity. S_R is a state update. Upon adaptation some values in the application state may require to be updated to reflect the change (see Example 6). V_R is the set of public variables expected by the new activity P_R. Finally, NF_R is the set of non-functional properties that will be provided by the new implementation P_R and CF_R the compulsory flag, specifying whether the update is compulsory or not.

Example 6. The Example 1 in the Introduction can be implemented by:

$$D_R, Number = IC2356 \vdash P_R(\{Number = FR82\},$$
$$\{Number, Source, Destination\}, \{Time = 83m, Cost = 32euros\}, false)$$

where the description D_R matches with D_A. Here the applicability condition c_R is just $Number = IC2356$, i.e. we assume to have such a rule for each train that could be replaced by a different connection, and that the train number is enough to identify it. If adaptation has to be performed, then the new code P_R will be installed, for booking and taking the FrecciaRossa train. In this case, the state will be updated by setting $Number$ to $FR82$, the number of the FrecciaRossa train (it is not enough to add an assignment to P_R, since in this last case the state update will be executed only when the new activity will be scheduled). The new activity will require to exploit the public variables $\{Number, Source, Destination\}$ and will guarantee as new non-functional properties $\{Time = 83m, Cost = 32euros\}$. Since the old train connection is still available, this update is not compulsory (i.e., the compulsory flag is set to false).

When Bob enters the train station and its Travelling application registers to the adaptation server of the train station (if the On Registration approach is used), or before the activity Take Train is started (if the On Activity Enter approach is used), or at some other point in time (depending on the used approach), the check for adaptation is performed according to the algorithm in Table 1. We show now how Algorithm 1 is applied to Example 1.

1. the description D_R in the rule is matched with the description $D_{TakeTrain}$ of the activity using function $MATCH$; the two description matches;
2. it is checked whether the public variables of the application are enough for running the new code, i.e. whether $V_R \subseteq V_A$;
3. the applicability condition $Number = IC2356$ is evaluated; this holds;
4. the compulsory flag CF_R is checked; we assume here that it is false, i.e. the old train connection is still available;
6. the non-functional properties of the new implementation are compared with the old ones, i.e. $COMP_{TakeTrain}(\langle 161, 11\rangle, \langle 83, 32\rangle)$ is computed; this evaluates to true.

Algorithm 1. Rule matching algorithm

Require: Activity definition $\langle D_A, V_A, NF_A, COMP_A \rangle$, rule definition $D_R, c_R \vdash (S_R, V_R, NF_R, CF_R)$, adaptation manager state V_ρ

```
 1: if MATCH(D_R, D_A) == true then
 2:   if V_R ⊆ V_A then
 3:     if c_R(V_A, V_ρ) then
 4:       if CF_R == true then
 5:         return true
 6:       else if COMP_A(NF_A, NF_R) == true then
 7:         return true
 8:       else
 9:         return false
10:       end if
11:     end if
12:   end if
13: end if
```

After these checks have been performed and succeeded, adaptation has to be performed. This requires the following steps:

1. the adaptation server sends the new code P_R to the application, which replaces the old code of the activity P_A;
2. the adaptation interface of the application is updated, with the new non-functional properties NF_R replacing the old non-functional properties NF_A;
3. the state of the application is updated by setting variable $Number$ to $FR82$, the number of the FrecciaRossa train.

The first step is the more tricky, since the new code P_R needs to be sent from the adaptation server to the application and integrated with the rest of the application. For instance, it should be able to exploit the public variables of the application. To show how this issue can be solved, and how the whole approach can be applied in practice in a service-oriented setting we move in the next section to a practical example.

4 Dynamic Adaptation in Service-Oriented Applications

In this section we describe JoRBA (Jolie Rule-Based Adaptation framework), a proof-of-concept implementation of our adaptation mechanisms based upon the Jolie (Java Orchestration Language Interpreter Engine) language. Jolie [16,11] is a full-fledged programming language based upon the service-oriented programming paradigm, suited for rapid prototyping of service-oriented applications.

The service-oriented paradigm offers an easy way to model loosely coupled interactions such as the ones between our adaptation manager and the adaptable applications interacting with it. As such, each adaptation server offers a set of public interfaces that the adaptable applications can exploit in order to check when and whether adaptation is needed and to apply it. We have chosen

Jolie since it offers native primitives that are based upon the service-oriented paradigm, which simplify the implementation of the complex interactions required by our approach. For instance, the use of *dynamic embedding* has been fundamental to perform the replacement of activities during adaptation. Embedding is a Jolie mechanism that allows for the creation of private instances of services. Jolie allows to embed new services at runtime.

Jolie is an open source project released under the LGPL license, whose reference implementation is an interpreter written in the Java language.

The Jolie language takes inspiration from both sequential languages and concurrent calculi. It includes in fact assignments, if-then-else, while and other statements with a syntax similar to those, e.g., of C and Java, but it also provides parallel composition as a native operator and allows message passing communications by means of its One-Way and Request-Response communication patterns, inspired by WSDL [24] and WS-BPEL [18]. Jolie allows to easily manipulate structured data such as trees and XML-like structures. In fact, Jolie variables are labelled trees, where nodes can be added and removed dynamically.

4.1 JoRBA Architecture

We describe now the overall architecture of JoRBA, the Jolie prototype implementing the approach for dynamic adaptation described in the paper. For simplicity, JoRBA is based on the **On Activity Enter** approach for triggering adaptations, and on the **Sequential** approach for rule order. JoRBA includes both an adaptation manager composed by different distributed adaptation servers and a general skeleton for adaptable applications (together with a sample instance). JoRBA and the implementation of the Travelling scenario are available at [12]. The overall architecture of JoRBA is represented in Fig. 1 using a Collaboration Diagram (see [6]) and described below.

The adaptation server is composed by two roles: *AdaptationManager* and *AdaptationServer*. An AdaptationServer handles a set of adaptation rules and their related functionalities. We allow for many AdaptationServer instances to run simultaneously coordinated by the AdaptationManager service. The AdaptationManager service is also responsible for managing a global state, including information on the environment conditions, and handling requests coming from the adaptable applications.

The adaptable applications are implemented as *Client* services, which implement the behavior of the application relying on two other services for managing the adaptation mechanisms:

- ActivityManager: handles the execution of adaptable activities; in particular it provides an operation `run` to execute an adaptable activity when requested by Client; since JoRBA is based on the **On Activity Enter** approach, before starting the execution of the activity the ActivityManager invokes the AdaptationManager service to look for updates;
- State: manages the state of the application and allows the adaptation manager (as well as the application code itself) for accessing it when needed.

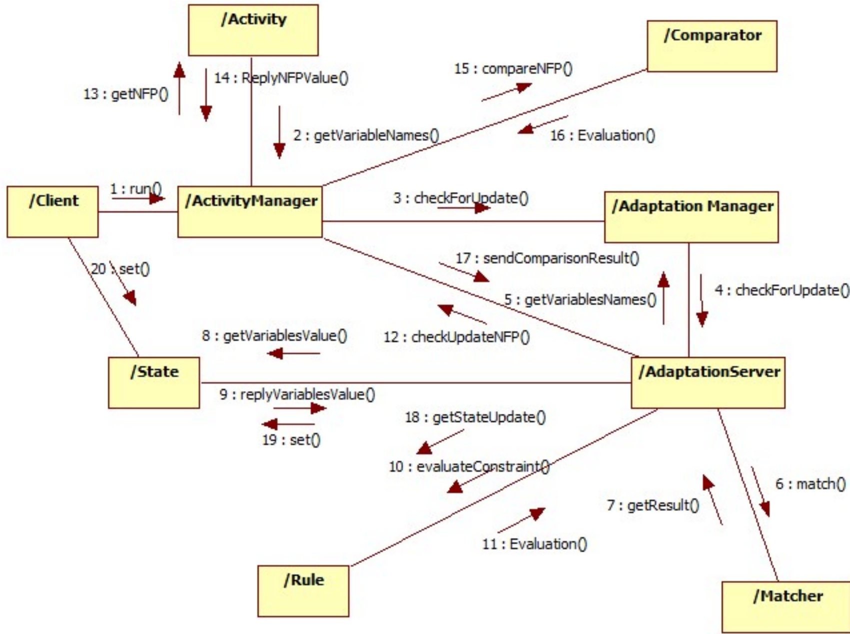

Fig. 1. Collaboration Diagram of JoRBA

Client services can easily be implemented by extending the *AbstractClient* definition, and complementing it with the user-defined behavior of the application. More specifically, the client application must include the `AbstractClient.iol` file, which in turn embeds the ActivityManager and State services. The code of the application should initialize the public variables by interacting with the private State service. Also, activities should be defined in separate files and executed by calling the `run` operation of the private ActivityManager service.

Whenever a Client service invokes the AdaptationManager service via the ActivityManager, the AdaptationManager queries all the registered AdaptationServer services in sequential order. When an AdaptationServer starts, it registers itself to the AdaptationManager service, and initialize itself. In particular, it scans its `rules` subdirectory for rule definitions. Each rule is defined as a service extending the `AbstractRule.iol` file. Each rule should define three procedures:

- `dataInit`: initializes the data structure with the information concerning the rule, including a reference to the new code for the activity;
- `onEvaluateConstraint`: implements the applicability condition c;
- `onGetStateUpdate`: specifies the state update for the client.

Upon invocation, each AdaptationServer service scans its rules in sequential order, checking if each of them is applicable using Algorithm 1. The implementation of this algorithm relies on Matcher, an internal auxiliary service that implements the *MATCH* function for comparing the activity description with

the corresponding description in the rule. In the current implementation the Matcher service just performs an equality check between the two descriptions, however one can easily refine it by implementing his preferred matching policy. The AdaptationServer interacts with the states of the AdaptationManager and of the Client to get the values necessary for checking the applicability conditions. It also interacts with the ActivityManager of the Client for checking if the non-functional properties provided by the new activity are better than the ones provided by the current activity, according to the user-specified policies (which are encoded in the Comparator service embedded by the ActivityManager). If all the checks succeed then the AdaptationServer updates the State of the Client with the new values specified by the adapted activity and, finally, sends back to the invoking ActivityManager the updated code for the activity. The latter is dynamically embedded by the ActivityManager, replacing thus the old code. A sample execution of the Travelling scenario implemented using JoRBA can be found in Appendix A.

5 Related Works and Conclusions

Most of the approaches to adaptation found in the literature concern built-in adaptation, i.e. the adaptation logic is completely specified at design time. They concentrate on how to specify adaptation mechanisms and adaptable applications, exploiting different tools. For instance, the specification may be performed by extending standard notations (such as BPEL [18]) with adaptation-specific tools [13], using event-condition-action like rules [2,7], variability modeling [10] or aspect-oriented approaches [14]. Other works extend Software Architectures [19] to deal with adaptation, giving rise to *Dynamic Software Architectures* (DSAs) [15,8]. Other approaches to built-in adaptation instead define novel languages to specify structural reconfiguration aspects [9,15,21], that have been proposed with the objective of architecture-based dynamic adaptations.

There is a main difference between the proposals listed above and ours, since their adaptation logics are hard-wired into the application and defined at design-time, while we separate the running application from the adaptation logic, allowing to create and update the latter after application deployment (i.e., at runtime).

In the literature there are however proposals of frameworks for dynamic adaptation, all featuring an adaptation manager separated from the application. We will compare with them below, considering the following aspects: *(i)* whether the set of adaptation rules can be created and modified during the execution of the application; *(ii)* whether the choice of which rule to apply is static or dynamic; *(iii)* whether adaptation is aimed at changing the functionalities of the application or *(iv)* optimizing the non-functional properties. The results of the comparison are depicted in Table 2. Notably, all the listed approaches are in the service-oriented field.

In [20] the authors consider the problem of adapting the application by replacing malfunctioning services at runtime. The adaptation rule is fixed at design time, but it is dynamically applied by a *manager* component that monitors

Table 2. Features of frameworks for dynamic adaptation

Framework	Dynamic adaptation rules	Dynamic rule selection	Functional improvement	Non-functional optimization
Spanoudakis et al.[20]	–	+	+	+
Narendra et al.[17]	–	+	–	+
METEOR-S[23]	–	–	–	+
PAWS[1]	–	+	+	+
Our framework	+	+	+	+

functional and non-functional properties, creates queries for discovering malfunctioning services and replaces them with dynamically discovered replacements.

[17] proposes an aspect-oriented approach for runtime optimization of non-functional QoS measures. Here aspects replace our adaptation rules. They are statically defined, but dynamically selected.

The METEOR-S framework [23] supports dynamic reconfiguration of processes, based on constraints referring to several QoS dimensions. Reconfiguration is performed essentially at deployment-time.

PAWS [1] is a framework for flexible and adaptive execution of web service-based applications. At design-time, flexibility is achieved through a number of mechanisms, i.e., identifying a set of candidate services for each process task, negotiating QoS, specifying quality constraints, and identifying mapping rules for invoking services with different interfaces. The runtime engine exploits the design-time mechanisms to support adaptation during process execution, in terms of selecting the best set of services to execute the process, reacting to a service failure, or preserving the execution when a context change occurs.

As can be seen, there is a lot of work on dynamic adaptation, but still lot of space for improvements. Some of our directions for future work have been already cited throughout the paper. For instance, we want to update running activities, preserving their state. This requires to put more information in the adaptation interface of applications. We also want to apply our approach outside the service-oriented area, where most of the approaches are, moving to the object-oriented paradigm. Finally, we want to define type systems on rules and on adaptable activities to guarantee that during all the evolution some basic properties (e.g., security, deadlock freeness,...) are preserved.

Acknowledgments. Authors thank the anonymous reviewers for valuable comments and suggestions.

References

1. Ardagna, D., Comuzzi, M., Mussi, E., Pernici, B., Plebani, P.: PAWS: A framework for executing adaptive web-service processes. IEEE Software 24(6), 39–46 (2007)
2. Baresi, L., Guinea, S., Pasquale, L.: Self-healing BPEL processes with Dynamo and the JBoss rule engine. In: Proc. of ESSPE 2007, pp. 11–20. ACM Press, New York (2007)

3. Brun, Y., et al.: Engineering self-adaptive systems through feedback loops. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) SESAS 2009. LNCS, vol. 5525, pp. 48–70. Springer, Heidelberg (2009)
4. Bucchiarone, A., et al.: Design for adaptation of service-based applications: Main issues and requirements. In: Proc. of WESOA 2009 (2009) (to appear)
5. Bucchiarone, A., Lluch Lafuente, A., Marconi, A., Pistore, M.: A formalisation of Adaptable Pervasive Flows. In: Proc. of WS-FM 2009 (2009) (to appear)
6. Bultan, T., Fu, X.: Specification of realizable service conversations using collaboration diagrams. Service Oriented Computing and Applications 2(1), 27–39 (2008)
7. Colombo, M., Di Nitto, E., Mauri, M.: SCENE: A service composition execution environment supporting dynamic changes disciplined through rules. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 191–202. Springer, Heidelberg (2006)
8. Floch, J., Hallsteinsen, S., Stav, E., Eliassen, F., Lund, K., Gjorven, E.: Using architecture models for runtime adaptability. IEEE Software 23(2), 62–70 (2006)
9. Garlan, D., Schmerl, B.: Model-based adaptation for self-healing systems. In: Proc. of WOSS 2002, pp. 27–32. ACM Press, New York (2002)
10. Hallerbach, A., Bauer, T., Reichert, M.: Managing process variants in the process life cycle. In: Proc. of ICEIS, vol. (3-2), pp. 154–161 (2008)
11. Jolie team. Jolie website, `http://www.jolie-lang.org/`
12. Jorba v0.1., `http://www.jolie-lang.org/examples/tgc10/JoRBAv0.1.zip`
13. Karastoyanova, D., Houspanossian, A., Cilia, M., Leymann, F., Buchmann, A.P.: Extending BPEL for run time adaptability. In: Proc. of EDOC 2005, pp. 15–26. IEEE Press, Los Alamitos (2005)
14. Kongdenfha, W., Saint-Paul, R., Benatallah, B., Casati, F.: An aspect-oriented framework for service adaptation. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 15–26. Springer, Heidelberg (2006)
15. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: Proc. of FOSE 2007, pp. 259–268 (2007)
16. Montesi, F., Guidi, C., Zavattaro, G.: Composing services with JOLIE. In: Proc. of ECOWS 2007, pp. 13–22. IEEE Press, Los Alamitos (2007)
17. Narendra, N.C., Ponnalagu, K., Krishnamurthy, J., Ramkumar, R.: Run-time adaptation of non-functional properties of composite web services using aspect-oriented programming. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 546–557. Springer, Heidelberg (2007)
18. OASIS. Web Services Business Process Execution Language Version 2.0., `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html`
19. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. SIGSOFT Softw. Eng. Notes 17(4), 40–52 (1992)
20. Spanoudakis, G., Zisman, A., Kozlenkov, A.: A service discovery framework for service centric systems. In: Proc. of SCC 2005, pp. 251–259. IEEE Press, Los Alamitos (2005)
21. Taylor, R.N., van der Hoek, A.: Software design and architecture: The once and future focus of software engineering. In: Proc. of FOSE 2007, pp. 226–243 (2007)
22. van Lamsweerde, A.: Requirements Engineering: From System Goals to UML Models to Software Specifications. Wiley, Chichester (2009)
23. Verma, K., Gomadam, K., Sheth, A.P., Miller, J.A., Wu, Z.: The meteor-s approach for configuring and executing dynamic web processes. Technical report, University of Georgia, Athens (2005)
24. World Wide Web Consortium. Web Services Description Language (WSDL) 1.1., `http://www.w3.org/TR/wsdl`

A The Travelling Scenario in Jolie

The JoRBA prototype [12] includes not only the basic services implementing the adaptation manager and the skeleton for adaptable applications described

Adaptation Server

```
Demo> jolie server2/AdaptationServer_2.ol
Loading rule TakeBusRule.ol
Loading rule TakeTrainRule.ol
Checking rule TakeBusRule.ol for activity GoToMeeting
        Descriptions MATCH not positive, skipping rule
Checking rule TakeTrainRule.ol for activity GoToMeeting
        Descriptions MATCH not positive, skipping rule
Checking rule TakeBusRule.ol for activity TakeTrain
        Descriptions MATCH not positive, skipping rule
Checking rule TakeTrainRule.ol for activity TakeTrain
        Descriptions MATCH positive
        Execution variables for new activity available
        Variables for constraint evaluation available
        Constraint satisfied
        Not compulsory rule, asking client for user preferences checking...
        Rule applies, sending new code
Checking rule TakeBusRule.ol for activity TakeBus
        Descriptions MATCH positive
        Execution variables for new activity available
        Variables for constraint evaluation available
        Constraint satisfied
        Not compulsory rule, asking client for user preferences checking...
        User preferences not met, skipping rule
Checking rule TakeTrainRule.ol for activity TakeBus
        Descriptions MATCH not positive, skipping rule
```

Adaptation Server

Client

```
Demo> jolie Client.ol
Client started
Entering activity GoToMeeting
        Asking the adaptation manager for updates
        Executing activity
Entering activity TakeTrain
        Asking the adaptation manager for updates
        Evaluating user preferences
        Update rejected; cause: NFP not satisfying user preferences
        Evaluating user preferences
        Update accepted
        Executing activity
New high speed train selected
Buying train ticket...
Entering activity TakeBus
        Asking the adaptation manager for updates
        Evaluating user preferences
        Update rejected; cause: NFP not satisfying user preferences
        Executing activity
Buying bus ticket...
```

Client

Fig. 2. Screenshot of prototype execution

in Section 4, but it also contains the sample Travelling application described in the examples presented throughout the paper and a few adaptation servers.

The main application has an adaptation interface including three different activities: the main activity Go To Meeting and the two subactivities Take Train and Take Bus. A sample execution is in Figure 2.

The client is executing in the bottom console. First the activity Go To Meeting is entered. The adaptation manager looks for updates, but there is no update matching this activity (as can be seen from the adaptation server console, in the top part of the figure). When the activity Take Train is started instead, two matching rules are found. All the checks are performed. The first one is discarded because of non-functional properties that do not satisfy user preferences. The second one instead is applied. Later on updates are checked also for activity TakeBus. The only update available is not applied because of the non-functional properties.