A Pattern-Based Architecture for Dynamically Adapting Business Processes

Mohamed Lamine Berkane[1] Lionel Seinturier[2] Mahmoud Boufaida[1]
[1] LIRE Laboratory
Mentouri University of Constantine, Algeria
{ml.berkane,mboufaida}@umc.edu.dz

[2]LIFL-INRIA ADAM
University of Lille, 59655 Villeneuve d'Ascq, France
Lionel.Seinturier@lifl.fr

Abstract— **The need to adapt a business process in applications has been a topic of interest in the recent years. Several approaches offer solutions to it. But, a limitation of most existing ones is the tight coupling of the adaptation logic with the execution one inside the engine implementation. In addition, they use the adaptation of business process only in the implementation phase (at runtime). To address these problems, we propose an architecture to develop a business process adaptation system. This architecture introduces modularity with an approach based on design patterns. We use some patterns to separate the adaptation logic and the functional one, and to address the adaptation at both the design phase and the implementation one. We show the feasibility of the proposed approach through the TRAP/BPEL framework.**

Keywords-Business Process; Design pattern; Abstraction Layers; Adaptation logic.

I. INTRODUCTION

Web services have evolved as a means to integrate processes and applications at an inter-enterprise level [17]. Several Web services can be combined to compose a new system. This last one can be seen as a composite Web service, which usually implements a business process.

A business process describes a sequence of tasks. Each task represents a coherent set of activities that fulfill a specific functionality. Tasks can be delegated to services and may require human interaction. Most business process languages assume that the tasks are executed in a static context. However, business process environments are often dynamic. For example, services can become unavailable, unexpected faults may occur or participating partners in the business process may not be known upfront, before some tasks are actually executed. In these situations, it is important to adapt a business process's behavior at run time in response to changing requirements and environmental conditions.

Recently, various approaches have proposed to support the dynamic business process adaptation: AO4BPEL (Aspect-Oriented for Business Process Execution Language) [1][2][16], VxBPEL [3], TRAP/BPEL (Transparent Reflective Aspect Programming/Business Process Execution Language) [4], CEVICHE (Complex EVent processIng for Context-adaptive processes in pervasive and Heterogeneous Environments) [5], MASC (Manageable and Adaptable Service Compositions) [6], DYNAMO (Dynamic Monitoring) [7], MVC (Model-View-Controller) [15]. However, most of these approaches do not treat changes at the design phase, and focus on run-time adaptation in terms of process instances. In addition to this, the current lack of reusable adaptation expertise can be leveraged from one adaptation system to another further exacerbates the problem.

In this paper, we present a pattern-based architecture for designing the adaptation system of business process. In our architecture, the system is designed in a modular way based on design patterns [8][9][10][12]. These patterns offer flexible solutions to common system development problems [12]. They express solutions of a known and recurrent problem in a particular context. Some of these patterns are used to specify the components of the adaptation systems. These components are: monitoring, decision-making, and reconfiguration [9][10]. Monitoring enables an adaptation system to aware the bussines process and detect conditions warranting reconfiguration, decision-making determines what set of monitored conditions should trigger a specific reconfiguration response, and reconfiguration enables an adaptation system to change the bussines process in order to fulfill the business requirements. Based on design patterns, our architecture supports the design of the adaptation system in four levels: the requirement layer, the functional layer, the logical layer and technical one. These abstraction layers are ordered hierarchically starting with (very abstract) high layers and leading to (very concrete) low layers. Each abstraction layer provides concepts for representation of the adaptation information, which is specific for each development phase. During the transition from a higher layer to a more concrete layer, the model information is enriched.

The rest of this paper is organized as follows: Section 2 presents some of the related work, Section 3 presents the proposed architecture and shows how it is realized using patterns; in Section 4, we use a case study to demonstrate the feasibility of our architecture through the TRAP/BPEL framework, and Section 5 concludes and discusses some future work.

II. RELATED WORK

This section overviews selected efforts conducted by researchers to facilitate the development of dynamically adapting business process system.

Adaptation and patterns: Ramirez and Cheng [8][9][10] presented several patterns for developing adaptive systems. These patterns are classified into three key elements of adaptive systems: monitoring, decision-making, and reconfiguration. The authors do not offer an approach to use these patterns. Beside, these patterns are generic; they can be used in adaptive systems as the multi-agent systems, network applications and information systems. In our case, we use some of these patterns to define aspects which are relevant for running the business process.

Gomaa et al. [14][19] proposed some patterns to specify the dynamic behavior of software architectures (master/slave, centralized, server/client, and decentralized architectures). These patterns are helpful to the developers implementing dynamically adaptation systems. Moreover, these approaches support only some kinds of software architectures, and the proposed patterns are specific to these architectures.

The GoF (Gang of Four) patterns [12] are the most popular and widely used in the designed system, and also are used in the abstract level. However, these patterns do not provide a solution to the adaptation problems. But, we use some of these patterns, to identify objects of the business process adaptation system at a high level of abstraction.

Adaptation and business process: Charfi et al. [16] presented a plug-in based architecture for self-adaptive processes that uses AO4BPEL [1]. Each plug-in has two types of Aspects: the monitoring Aspects that will check the system to observe when an adaptation is needed and the adaptation Aspects that will handle the situations detected by the monitoring Aspects. Yet, this approach supports only two kinds of components (Aspects) to adapt the business process. However, this approach defines the adaption logic at run time, while in our approach, the adaption logic is defined both at design-time and at runtime. In addition, our approach defines three components to separate the functional logic from the adaptive one. This makes our approach more modular.

Koning et al. [3] presented a language, called VxBPEL. They extended the BPEL language to add new elements like VariationPoint and Variant to capture variability in a service-based system. The first element specifies the places where the process can be adapted, and the second define the alternative steps of the process that can be used. This approach defines the adaptive logic both at design-time and at runtime. Yet this approach defines a new language to support the adaptation, and extends an existing engine to support VxBPEL language. In our case we use the standard BPEL, and we keep the known process engine.

The work which is closer to our proposal is the one presented in [15]. The authors present a framework based on the Model-View-Controller (MVC) pattern to support the adaptation of BPEL processes in a dynamic and modular way. In this framework, a workflow process is designed as a template, where the tasks can be specified in an abstract level. Concrete implementations of the tasks, modeled as aspects, are then selected from a library according to policy-based adaptation logic. However, this approach uses the pattern notion (MVC) to support the adaptation of business processes, while in our approach, we use the pattern (MDR) to develop the adaptation system of business process. This makes our approach more generic.

Hermosillo et al. [5] present CEVICHE, a framework that combines Complex Event Processing (CEP) and Aspect Oriented Programming (AOP) to support dynamically adaptable business processes. The adaptation logic is defined as aspects (reconfiguration component), and adaptation situations are specified by CEP rules (monitoring component). However, the decision- making is not specified as component in this framework. It is integrate into the defined aspects.

Xiao et al. [18] propose a constraint-based framework for supporting dynamic business process adaptation. In this framework, process adaptations are performed in a modular way based on process fragments. Process fragments are standalone fragments of processes that can be reused across multiple processes. This approach separates between the functional logic and the adaptive one by using the process fragments. However, this framework presents the adaptation only at run-time; in addition it cannot apply changes to living process instances. When new process schemas are (re)generated, only new process instances will be created according to the new process schemas.

III. A LAYER-BASED ARCHITECTURAL MODEL FOR BUSINESS PROCESS ADAPTATION

In this section, we present a pattern-based architecture that permits the design of adaptation systems in a dynamic and modular way. This architecture is composed of four layers: the requirement layer, the functional layer, the logical layer, and the technical one. Each layer contains three components (except requirement level): monitoring, decision-making, and reconfiguration. These three components will be refined in three layers. The starting point is the requirement layer which is a set of requirements for a behavior of the adaptation system. These requirements can have different forms, for example the form of a textual documentation or a collection of Use Cases. Secondly, the functional layer provides the definition of the adaptation system's interfaces with business process. Thirdly, the logical layer provides an architectural view of the system by partitioning it into logical communicating components. It defines the total behavior of the system. Lastly, the technical layer represents the lowest level of abstraction. It focuses on aspects relevant for running the business process. This architecture can provide an appropriate level of abstraction to describe dynamic change in a business process, such as the use of components, rather than at the algorithmic level (Figure 2). In the proposed architecture, we focus on the functional, logical and technical layers.

A. Requirement Layer

The proposed approach imposes a clear separation of concerns between functional and adaptation requirements. The adaptation requirements are concerned with understanding how a system may either make a transition

between satisfying different functional requirements depending on context, or continue to satisfy the same functional requirements in the face of changing context. Hence, the adaptation requirements are intimately related to, and derived from, the functional requirements. These requirements can have different forms, for example the form of a textual documentation or a collection of Use Cases.

B. *Functional Layer*

In this layer, we provide the main functions of our adaptation system. It comprises the definition of its interfaces with business processes. Our adaptation system contains three main functions: monitoring, decision-making, and reconfiguration; and also two interfaces: to monitor and to reconfigure the business process. These functions can be seen as components, which communicate between them to adapt a business process's behavior in response to changing requirements and environmental conditions.

C. *Logical Layer*

In this layer, we refine the adaptation system presented in the previous sub-section. It defines three components specified by the functional level: monitoring, decision-making, and reconfiguration. In addition, there are two relationships: one between monitoring and decision-making, and the second between decision-making and reconfiguration, as shown in Figure 3. The logical components can be obtained as the combination of sub-components (objects) with respect to the dependencies between them. In this level, we use the GoF design patterns [12] to model the components and the sub-components. These patterns were chosen because they define the abstract concepts of adaptation (such as different strategies of adaptation defined by "Strategy" pattern).

The first component is the monitoring. The main objective of monitoring component is to enable an adaptation system to observe business process and environmental conditions that may warrant reconfiguration. To monitor the business process in the logical layer, we use the Observer pattern [12], which uses Observer and Subject objects. The Observer object collects the information about the business process and its environment, and the subject object is used to represent any component that needs to perform monitoring in business process. This last object defines the detection conditions to specify the conditions that may warrant reconfiguration (Figure 1). When a detection condition is detected, the subject objects notifies the observer objects, which in turn notifies the corresponding decision-making component.

The first relationship between monitoring component and decision-making one is defined to permit the interactions between the objects defined in the first component and the objects defined in the second one. We can have multiple interactions between the objects defined in these components. An object of the monitoring component can communicate with multiple objects of the decisions making component. Thus, an object of decision-making component may receive several messages from objects of the monitoring component. To carry the number of interaction between these two types of objects, it becomes necessary to define an intermediate object that manages these interactions. The 'Mediator' pattern [12] responds in a good way to this situation. By applying this pattern, the monitoring and the decision-making components can be evolving independently.

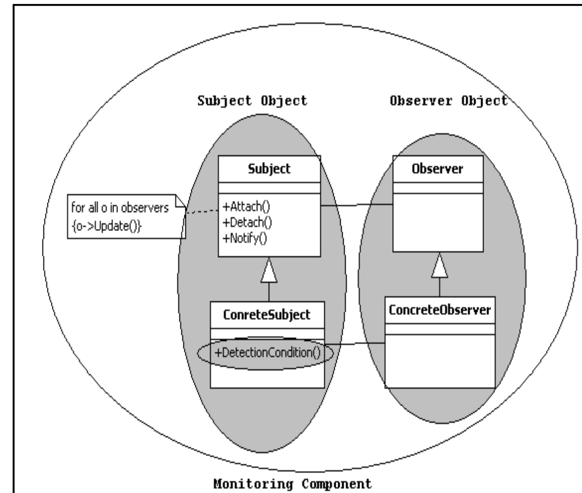

Figure 1. Monitoring Component in the logical layer

The second component (Decision-making) is the most important in the proposed architecture. The main objective of decision-making component is to determine when and how to reconfigure a business process in response to monitoring information. In this component, we define a family of algorithms. These one leverage a knowledge repository that associates specific monitoring scenarios with series of reconfiguration instructions. To define these algorithms, we use the 'Strategy' pattern [12]; this one creates a set of algorithms defined in the objects. Applying this pattern separates the functional logic from the decision-making one, thus clustering the set of reconfiguration responses for distinct events.

The second relationship between decision-making component and reconfiguration one is similar to the relationship between monitoring component and decision-making one, unless it manages the interactions between the objects of decision-making component and the objects of the reconfiguration one.

The last component (Reconfiguration) specifies in detail the actions defined in the algorithms of decision-making component. In this component, we use the 'Bridge' pattern [12] to separate between the algorithms (Abstraction) and the reconfiguration instructions (Implementor). In this pattern, we specify two kinds of objects: the Abstraction, and the Implementor. By applying this pattern, we can extend the Abstraction and the Implementor hierarchies independently.

D. *Technical layer*

This layer defines the aspects relevant for running the adaptation system. This layer also explains how the process adaptation is realized. Thereby, we use the design patterns defined for developing dynamically adaptation systems

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-221-9

[8][9][10]. These patterns were chosen because they use the platform-independent models to represent the adaptation solution.

A business process executes a series of activities in a sequence. It can be designed to show the sequence of service invocation. A Web Service is one of many types of services that a process can invoke. In general, the Web services are distributed application components that are externally available, and each Web service provides an interface that can be used to exchange the required information with business processes. In the monitoring component of this layer, we use the sensor factory pattern [8][9]; this pattern may be used when components (Web services) are distributed and each component (Web service) provides an interface that can be probed for the required information. In this pattern, a 'SimpleSensor' object can be used to sensor the component that needs to be monitored in business process. It replaces the Observer object of the 'Observer' pattern. In addition the methods 'Attach' and 'Detach' of 'ConcreteSubject' of Observer pattern were realized by 'SensorFactory', 'ResourceManager' and 'Registry' objects. The first object manages the addition and the removal of sensors in the business process, and the Clients interact with this object in order to gain access to a sensor. The second object determines if an existing sensor can be shared with one or more clients, and also, determines if the business process has enough resources to deploy a new sensor. The last object is responsible for tracking deployed sensors across the business process.

In the first relationship between monitoring component and decision-making one, we use the 'Content-based Routing' pattern [8][10]. This pattern should be applied when multiple clients require access to the same monitoring information. In our case, may be multiple monitoring components need access to the same decision-making component.

In the logical layer, we have used the strategy pattern to define a family of algorithms for decision-making components. In the technical layer, we use a 'Case-based Reasoning'pattern [8][10] to select the specific reconfigurations, and show how the reconfigurations can be executed at run time. The 'Case-based Reasoning' pattern can be applied when runtime scenarios that require reconfiguration can be reliably identified. The important objects of this pattern are: 'Trigger', 'Inference Engine', 'Decision', and 'Fixed Rules'. The 'Trigger' object contains relevant information about what caused the adaptation request. It should at least provide information about the error source, and the type of error observed. The 'Inference Engine' object is responsible for applying a set of 'Rules' to produce an action in the form of a 'Decision'. The 'Decision' object represents a reconfiguration plan that will yield the desired behavior in the system. The 'Fixed Rules' object contains a collection of 'Rules' that guide the 'Inference Engine' in producing a 'Decision'. These 'Fixed Rules' replace the strategy objects of the 'Strategy' pattern.

The second relationship between decision-making component and reconfiguration one is specified by the 'Divide and Conquer' pattern [8][10][20]. This pattern avoids potential business process inconsistencies, because the business process may require applying multiple reconfigurations in succession. The 'Divide and Conquer' pattern decomposes a complex reconfiguration into simpler reconfigurations, and it determines dependency relations between different reconfigurations. The 'Divide-and-Conquer' strategy is employed in many complex algorithms. With this strategy, a problem is solved by splitting it into a number of smaller sub-problems, solving them independently, and merging the sub-solutions into a solution for the whole problem. Conceptually, this pattern follows a straightforward approach. One task splits the problem, then forks new tasks to compute the sub-problems, waits until the sub-problems are computed, and then joins with the subtasks to merge the results.

The reconfiguration component uses two kinds of patterns 'Component Insertion' and 'Component Removal' [8][10]. In our case, we insert and remove the web service. For example, 'Component Insertion' pattern can be used to safely insert a new component (web service) at run time. The important objects of this pattern are: 'Adaptation Driver', 'Change Manager', 'Reconfiguration Plan', and 'Reconfiguration Rules'. The first object is responsible for ensuring that incoming Client requests are queued for further processing. The second object provides support for loading and unloading Components (web services) and their interconnections. The third object stores the specific sequence of instructions for reconfiguring the system at run time. This object replaces the 'Abstraction' object of the 'Bridge' pattern. The last object contains rules and instructions for specifying how basic reconfiguration operations are carried out in system. Some basic reconfiguration operations include Component insertion, removal, and swapping. This object replaces the 'Implementor' object of the 'Bridge' pattern.

IV. A CASE STUDY:"TRAP/BPEL FRAMEWORK"

In this section, a case study is used to demonstrate the feasibility of our approach. For this case, we have selected the TRAP/BPEL framework [4][13]. In our paper, we focus on an architectural approach not because the TRAP/BPEL framework is uninteresting or less promising, but we argue that the architectural level shows how the adaptation system components of this framework are separated and generality deals with the challenges posed. TRAP/BPEL is a framework that adds the autonomic behavior into existing BPEL processes. It aims to make an aggregate web service continue its function even after one or more of its constituent Web services have failed, and also adds the autonomic behavior to BPEL processes by using a generic proxy as an indirection layer to interact with the partner services. The generic proxy has a standard interface and works for all partner services of one or more adapt-ready BPEL processes. A recovery policy is used in the proxy to dictate the adaptation behavior for each monitored service [4]. This generic proxy can be reused for any BPEL processes. Therefore, it is possible to provide a common autonomic behavior to a set of services. Furthermore, an adapt-ready

process monitors the behavior of Web service partners and tries to tolerate their failure by forwarding the failed request to its generic proxy, which in turn will find an equivalent service to substitute the failed one [4].

The main components of this framework are: adapt-ready process and generic proxy. The first one represents the monitoring component, and the second specifies in both the decision-making and the reconfiguration components (Figure 4 (a)).

In the monitoring component of the logical layer, we use the 'Observer' pattern to specify the adapt-ready process. This process (Observer object) monitors the behavior of Web service partners (Subject object). If one of the partner services fails (Detection condition) then the adapt-ready process forwards the failed request to its proxy. The proxy is generated specifically for this adapt-ready process and provides the same port types as those of the monitored Web services. This port is the mediator (Mediator object) between the adapt-ready process and the generic proxy. In addition, the generic proxy can provide behavior either common to all adapt-ready BPEL processes or specific to each monitored invocation using some high level policies. It may take one of the following actions according to the policy: invoke the service being recommended in the policy; find and invoke another service to substitute for the monitored service, or retry the invocation of the monitored service in the event of its failure. These three policies are considered as the different strategies (of 'Strategy' pattern) of decision-making component. The 'Bridge' pattern is responsible for the management of the policies (Abstraction object); it concretizes the actions defined in the policies (Implementor object). In this framework, there is not a mediator between decision-making component and reconfiguration one, because the proxy plays two roles at the same time (Figure 4 (b)).

In the technical layer, the TRAP/BPEL framework needs to incorporate some generic hooks (sensors of 'Sensor Factory' pattern) at sensitive joinpoints in the BPEL process (i.e., the invoke instructions). These joinpoints are points in the execution path of the program at which adaptation code can be introduced at run time. The operations and input/output variables of the proxy are the same as those of the monitored invocations. When more than one service is monitored within a BPEL process, the interface for the specific proxy is an aggregation of all the interfaces of the monitored Web services; this situation is specified by 'Content-based Routing' pattern. This last one defines an architecture (many-to-one) that gathers data from the different web services (one or more adapt-ready BPEL processes (multiple monitoring components)) and distributes it to the specific proxy (one decision-making component).

The proxy uses 'Case-based Reasoning' pattern to specify the behavior of decision-making component in the technical layer. This proxy (Inference Engine object) checks all the intercepted invocations (Trigger object) and tries to match these invocations with the specified policies (Fixed Rules object). If it finds a policy for that invocation, the proxy behaves accordingly to that, it selects one of three actions (Rule object); otherwise it follows its default behavior (Figure 4 (c)).

We use the 'Component Insertion' pattern to define the behavior of reconfiguration component. This pattern inserts a new web service (i.e. invoke a new web service). In the generic proxy, we cannot establish the relationship between decision-making component and reconfiguration one because it defines the behavior of two components (decision-making and reconfiguration) in one component.

Our Pattern-Based Architecture approach has several advantages over a framework-oriented approach (like TRAP/BPEL, AO4BPEL, CHEVICHE, etc) at developing dynamically adapting business processes. The design patterns provide general models that need to be instantiated and customized before they are implemented. Since models operate at a higher-level of abstraction than frameworks, they impose fewer initial constraints upon the system being developed. In addition, with our design pattern approach, developers select only those adaptation mechanisms their application will require. In contrast, adaptation-oriented frameworks provide infrastructure to perform the adaptation tasks for a wide range of applications; the overall infrastructure is needed for the adaptive application, even if not all the features are needed or used.

V. CONCLUSION AND FUTURE WORK

In this paper, we have presented a pattern-based architecture for designing the adaptation system of business processes. The proposed architecture is composed of four layers: the requirement layer, the functional layer, the logical layer, and the technical one. Then, in each layer, we defined three components: monitoring, decision-making, and reconfiguration. Finally, in each component, and for each layer, we use patterns to facilitate the reuse of adaptation expertise. These patterns separate the adaptation logic from the functional one. This separation of concerns facilitates the reuse of adaptation designs across multiple adaptation systems.

In the future, we will try to propose a hybrid approach that allows the use of the various adaptation components (monitoring, decision-making, and reconfiguration) of the different adaptation approaches (like AO4BPEL, CEVICHE, etc).

REFERENCES

[1] A. Charfi and M. Mezini. "Aspect-oriented web service composition with AO4BPEL". In Proceedings of the 2nd European Conference on Web Services (ECOWS), volume 3250 of LNCS, pp. 168–182. Springer, September 2004.

[2] A. Charfi, B. Schmeling, A. Heizenreder, and M. Mezini. "Reliable, secure, and transacted web service compositions with ao4bpel". In Proceedings of the 4th IEEE European Conference on Web Services(ECOWS), December 2006.

[3] M. Koning, C.-a. Sun, M. Sinnema, and P. Avgeriou, "Vxbpel:Supporting variability for web services in bpel," Inf. Softw.Technol., vol. 51, no. 2, pp. 258–269, 2009.

[4] O. Ezenwoye, S. Sadjadi, “TRAP/BPEL: A Framework for Dynamic Adaptation of Composite Services”, Tech. Rep. FIU-SCIS-2006-06-02, School of Computing and Information Sciences, Florida International University, 2006.

[5] G. Hermosillo, L. Seinturier, L. Duchien, “Using Complex Event Processing for Dynamic Business Process Adaptation” in Proceedings of the 7th IEEE 2010 International Conference on Services Computing (SCC 2010), Miami, Florida : United States, 2010.

[6] A. Erradi, V. Tosic, and P. Maheshwari. “Masc - .netbased middleware for adaptive composite web services”. In ICWS International Conference on Web Services, pages 727–734. IEEE Computer Society, 2007.

[7] L. Baresi and S.Guinea. “Dynamo and self-healing bpel compositions”. In ICSE COMPANION ’07 : Companion to the proceedings of the 29th International Conference on Software Engineering, pages 69–70, Washington, DC, USA, 2007. IEEE Computer Society, 2007.

[8] A. J. Ramirez. Design patterns for developing dynamically adaptive systems. Master's thesis, Michigan State University, East Lansing, MI 48823, 2008.

[9] A. J. Ramirez and B. H. C. Cheng. “Developing and applying design patterns for dynamically adaptive systems”. In 6th IEEE International Conference on Autonomic Computing, ICAC ’09 Barcelona, Spain, 2009.

[10] A. J. Ramirez and B. H. C. Cheng. “Developing and applying design patterns for dynamically adaptive systems”. In 5th International Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS'10, Cape Town, South Africa, May, 2010.

[11] A. Campetelli, M. Feilkas, M. Fritzsche, A. Harhurin, J. Hartmann, M. Hermannsdorfer, F. Holzl, S. Merenda, D. Ratiu, B. Schatz, and W. Schwitzer. “Model-based development – motivation and mission statement of workpackage zp-ap 1”. Technical report, Technische Universität München, 2009.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. “Design Patterns: Elements of Reusable Object-Oriented Software”. Addison-Wesley Professional, 1995.

[13] F. Baligand “Une Approche Déclarative pour la Gestion de la Qualité de Service dans les Compositions de Services “, Doctorate’thesis l’Ecole des Mines de Paris, 2008.

[14] H. Gomaa and M. Hussein. “Software reconfiguration patterns for dynamic evolution of software architectures“. In WICSA’04: Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture, page 79, Washington, DC, USA, 2004.

[15] K. Geebelen, E. Kulikowski, E. Truyen and W. Joosen “A MVC Framework for Policy-Based Adaptation of Workflow Processes: A Case Study On Confidentiality” In 2010 IEEE International Conference on Web Services, 2010.

[16] A. Charfi, T. Dinkelaker, and M. Mezini, “A Plug-in Architecture for Self-Adaptive Web Service Compositions”, in the Proceedings of IEEE International Conference on Web Services (ICWS’09), pp. 35- 42, 2009.

[17] M. Little, Transactions and web services, Communications of the ACM 46 (10) (2003) 49–54, 2003.

[18] Z. Xiao, D. Cao, C. You and H. Mei,”Towards a Constraint-based Framework for Dynamic Business Process Adaptation” In 2011 IEEE International Conference on Services Computing, 2011.

[19] H. Gomaa “Pattern-based Software Design and Adaptation“. In PATTERNS 2011: Proceedings of the Third International Conferences on Pervasive Patterns and Applications, page 90-95, Roma, Italy, 2011.

[20] G. Mattson Timoth, A. Sanders Beverly, L. Massingill Berna. « Patterns for Parallel Programming ». Addison-Wesley Professional, 2004.

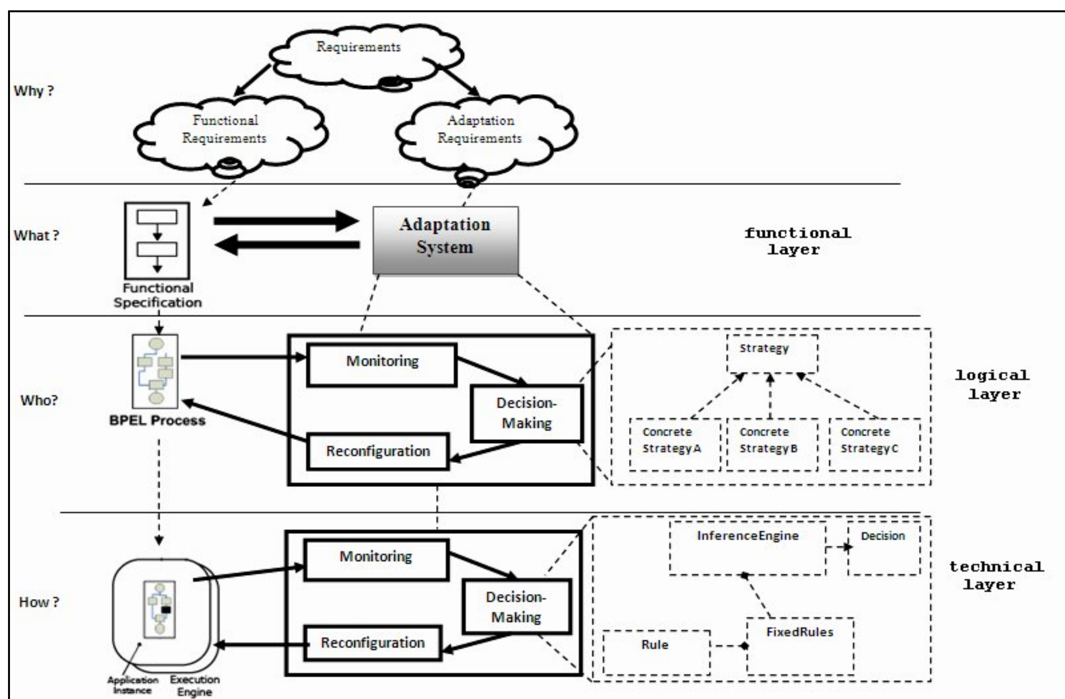

Figure 2. Overview of the proposed architecture

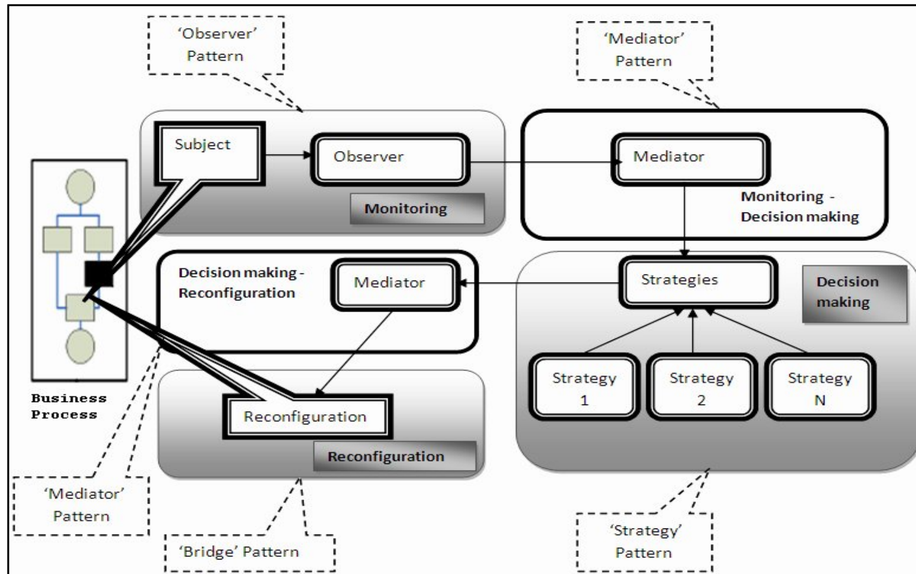

Figure 3. Representation of the logical layer

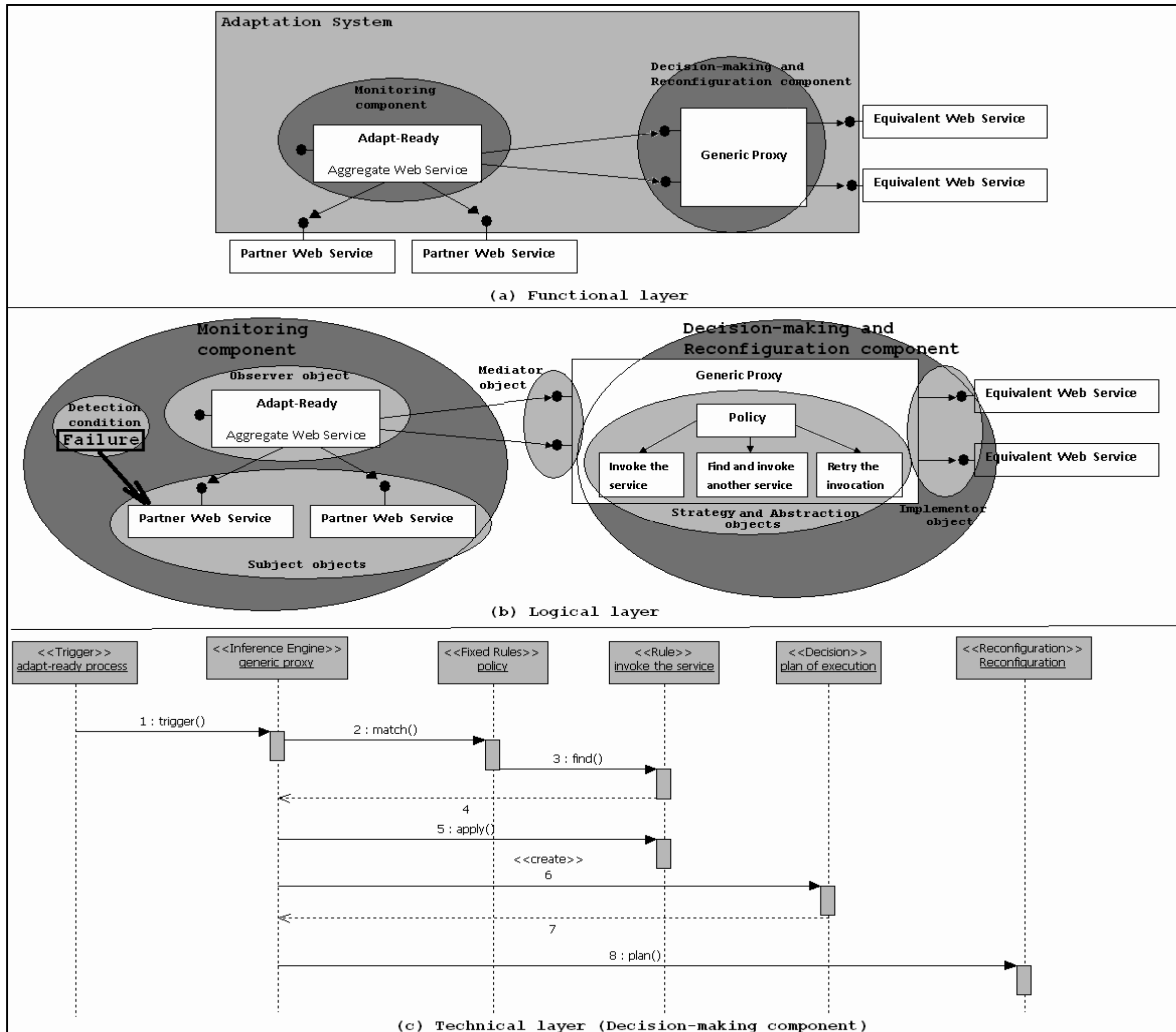

Figure 4. TRAP/BPEL framework in three layers