Electronic Communications of the EASST
Volume 43 (2011)

Proceedings of the
4th International DisCoTec Workshop on
Context-aware Adaption Mechanisms for Pervasive
And Ubiquitous Services
(CAMPUS 2011)

Towards a Flexible and Evolvable Framework for Self-Adaptation

Lucas Provensi and Frank Eliassen

6 pages

Guest Editors: Gabriel Hermosillo, Russell Nzekwa, Michael Wagner
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Towards a Flexible and Evolvable Framework for Self-Adaptation

Lucas Provensi[1] **and Frank Eliassen**[2]

[1] provensi@ifi.uio.no
[2] frank@ifi.uio.no
Department of Informatics, University of Oslo, Norway

Abstract: The growing complexity, scale and heterogeneity of software systems boosted a great deal of research in the field of self-management and self-adaptation. In general, current solutions are built as fixed frameworks, with rigid methodology, models and tools that are best suited for their target application domain but can not be easily applied in different domains. Furthermore, they lack the flexibility to let the developer make decisions on how the adaptation engine should work and do not consider the engine itself as a system subject to adaptation that can dynamically evolve. In this work-in-progress paper we discuss the requirements of a more flexible and evolvable framework for self-adaptation. We propose a conceptual model for realizing this framework, showing its benefits with an application scenario.

Keywords: self-adaptive systems, engineering, software evolution, control loop

1 Introduction

Self-adaptive or autonomous systems are software systems that present one or more self-* properties (self-healing, self-optimization, self-protection). Those systems are structured as closed-loops (often called *adaptation loop* or *adaptation engine*), which consist of a set of *control tasks* sharing a *knowledge base* and interacting to achieve the goal of the self-* properties. The system manages itself by continuously *monitoring* its internal state and external environment, *analyzing* the data to detect undesired operational states, *planning* how to adapt the system and *executing* the adaptation plan [KC03]. Despite the amount of work being done in the last years, there are still many challenges involving the engineering of self-adaptive software systems [ST09], mostly due to the difficulty of reusing the methods, techniques and tools offered by current frameworks in different and ever-evolving application domains.

When designing a self-adaptive system, developers usually start with partial knowledge of the application domain and are only able to elicitate an incomplete set of adaptation requirements. Consequently, as the developer learns more about the application domain, he needs to modify (refine, enhance or correct) the self-adaptive behavior. Modifying the behavior is not only constrained to the specification of a new set of adaptation goals, in some cases it may involve changing the adaptation engine itself. Monolithic and rigid frameworks can become an obstacle if, in a later development iteration, the set of adaptation methods, techniques and tools can not satisfy new adaptation requirements. We refer to *flexibility* as how easily the framework can be iteratively and incrementally modified to satisfy the needs of different stakeholders and reused in different application domains.

The flexibility has to do with static modifications, but sometimes these modifications need to be dynamic. Consider, for instance, the monitoring task of the adaptation loop. It shares resources (e.g. CPU cycles, network bandwidth) with the monitored system. Increasing the amount of monitoring can reduce the availability of resources to the monitored system but, on the other hand, decreasing it may imply in a less accurate and responsive adaptation. The problem of finding the optimal amount of monitoring can not be solved statically, because it depends on the dynamic conditions of the combined adaptation and adapted systems [BCNR10]. We use the term *evolvability*[1] as the ability of the framework's adaptation engine to dynamically reason about its operation and adapt itself according to new situations and requirements.

In this paper we present the requirements and a high-level description of what we believe would be a more flexible and evolvable framework for self-adaptation. The rest of the paper is structured as follows. In Section 2 we discuss the flexibility and evolvability requirements of such a framework. Section 3 presents a conceptual model and discusses its implementation as an adaptation middleware. Section 4 discusses the framework benefits using an application scenario. Section 5 discusses related works. Finally, Section 6 concludes the paper and present future works.

2 Requirements

One of the most advocated design principles for self-adaptive systems is the separation of concerns between the application logic and the self-adaptive behavior. Although this principle is related to reusability, in most frameworks it is realized as a monolithic external controller implementing the adaptation logic, that can be reused in similar application domains. We argue that a more fine grained separation of concerns is needed, changing the unit of reusability from the adaptation loop as a whole to its individual *control tasks*. This idea is in conformance with related works on the importance of making the control aspects of self-adaptive systems (control tasks and their relationships) explicit during design and clearly traceable in the implementation [MPS08]. We can synthesize this discussion in the following flexibility requirements:

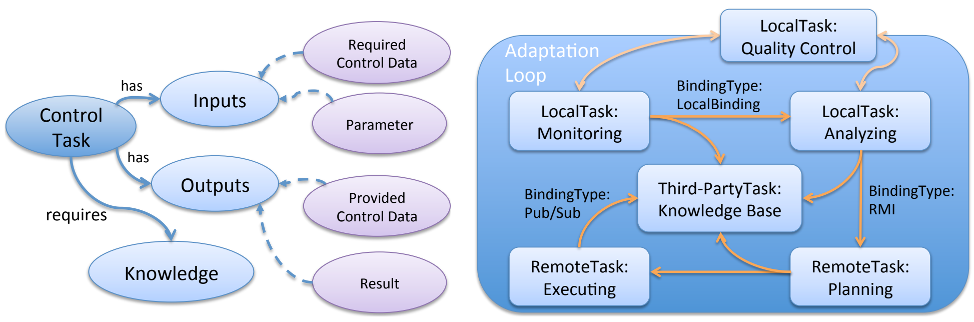
FR *- The adaptation engine should be designed as a closed-loop formed by a set of loosely coupled control tasks, giving the developer the flexibility to* ***1)*** *Specify the self-adaptive behavior using his current knowledge and expertise and* ***2)*** *Easily change the self-adaptive behavior and methodology*[2] *during the development life cycle, to reflect new knowledge and requirements.*

As discussed in Section 1, some characteristics of the self-adaptive behavior have important effects on the adapted system that can not be controlled statically. When the system is subjected to new operating states, the qualitative properties of the self-adaptive behavior, such as cost, safety and performance, can deviate significantly from the ideal and expected behavior [GEA06]. This uncertainty leads to the following evolvability requirements:

ER *- The framework should provide the means to* ***1)*** *Reason about the qualitative properties of the adaptation loop as a whole and of its control tasks individually and* ***2)*** *Modify the adaptation loop dynamically according to new requirements or changes in its operating environment.*

[1] The term evolvability was taken from the evolutionary biology and is defined as the ability of a population to generate and use genetic variations to respond to natural selection.

[2] Methodology is used in this paper as the set of methods applied to describe and realize the self-adaptive behavior.



(a) Control task (b) Adaptation Loop Specification (ALS)

Figure 1: Abstract description of a control task (a) and an ALS example (b)

3 Description of the Framework

In this section we present an abstract description of our framework. Subsection 3.1 introduces its conceptual model and shows how it fulfills the requirements presented in Section 2. In Subsection 3.2, we show how the framework can be implemented as an adaptation middleware.

3.1 Conceptual Model

The framework is based on exploiting a high-level specification of the self-adaptive behavior to dynamically construct the adaptation engine and manage its evolution. Since the unit of reusability is a control task, it is also the key concept in the design and the target of the engine self-management. Figure 1(a) shows the control task abstraction. The *required* and *provided control data* define the abstract type of the information consumed and produced by the task as part of an adaptation loop (e.g. the analyzing task consumes monitored data and produces symptoms). The *result output* refers to measurements (e.g. response time and memory footprint) and operation results (e.g. exceptions) exposed by the task. The *parameter input* refers to parameters used to adjust the task operation (e.g. sampling rate and time out). The *required knowledge* defines what type of human-defined domain knowledge (e.g. utility functions and quality models) the task works on (the knowledge itself can be described using independent languages).

The adaptation engine/loop in turn, is described as a set of interacting control tasks, as shown in Figure 1(b). This *adaptation loop specification* (ALS) is the main input used by the framework, driving the construction of the adaptation engine at load time and its evolution at runtime. The control tasks in the ALS are decoupled and use an open binding interaction model. The tasks can be implemented as different computational elements (objects, components or third-party services) and independently deployed by the framework locally or remotely. The binding type maps the abstract interaction to concrete operations of the chosen communication model (local interface binding, RMI, publish/subscribe).

The requirements described in Section 2 are satisfied by the model as follows. The requirement **FR-1** is satisfied by making the required knowledge explicit, giving the developers the flexibility to choose, according to their knowledge and expertise, what control tasks to use in the ALS.

To meet the requirement **FR-2**, at any development iteration, the self-adaptive behavior and methodology can be changed, by respectively modifying the knowledge used by the tasks and redesigning the ALS with new control tasks and knowledge types.

To fulfill the requirement **ER-1**, the result outputs of the control tasks can be used to calculate the deviation of the current self-adaptive behavior (based on runtime measurements) from the expected and ideal behavior (statically defined). To satisfy the requirement **ER-2**, the ALS can be further extended with tasks created to control the main adaptation loop, as the *Quality Control* task shown in Figure 1(b). These tasks can be implemented as simple controllers or as a complex adaptation sub-systems, consuming the result outputs from other tasks and controlling their behavior through their set of input parameters.

3.2 Middleware Implementation

The ALS is a high-level description of the adaptation process. We have considered a number of description languages, including industry trends, such as BPEL [3] and BPMN [4], and also more academic solutions, such as OWL-S [MBM$^+$07]. Those languages can be used to both describe and execute complex business processes. From our preliminary evaluation, OWL-S is a more adequate solution considering its full support to ontologies and flexible grounding model (supporting different interaction models). Using OWL-S API for Java [5], we were able to define and execute a simple ALS using the standard service ontology offered by OWL-S. Later this ontology can be extended to resemble the conceptual model introduced in Section 3.1.

A process execution engine, able to read and execute ALS, can be introduced as an extension of an adaptation middleware, that can leverage its services, such as factories and registries, to construct and execute the adaptation loop, as shown in Figure 2: First, the ALS and its required knowledge are deployed into the middleware(1). Then, an adaptation manager parses the ALS and starts the construction of the adaptation engine. The manager searches in a task registry for third-party tasks matching the ones described in the ALS (2). In this example, it finds a matching task (*Knowledge Base*) that supports the specified binding type (Pub/Sub). The required knowledge is stored in the knowledge base (3). The specification of remote tasks is dispatched to remote instances of the middleware (4). Finally, the managers in each middleware instance creates the appropriate tasks and connects them using the specified binding type (5).

4 Application Scenario

To demonstrate the flexibility and evolvability of our framework we use a hypothetical scenario of a multimedia application, where remote users can interact through real-time media streams. For simplicity, we focus only on control aspects of the application and abstract implementation details. We also assume that the developer has access to limited monitored information (e.g. network latency, bandwidth availability, CPU usage and battery level) and has control over a limited set of properties (e.g. the media temporal, spatial and qualitative dimensions).

[3] Web Services Business Process Execution Language - http://www.oasis-open.org/committees/wsbpel
[4] Business Process Modeling Notation - http://www.bpmn.org/
[5] OWL-S API - http://on.cs.unibas.ch/owls-api/

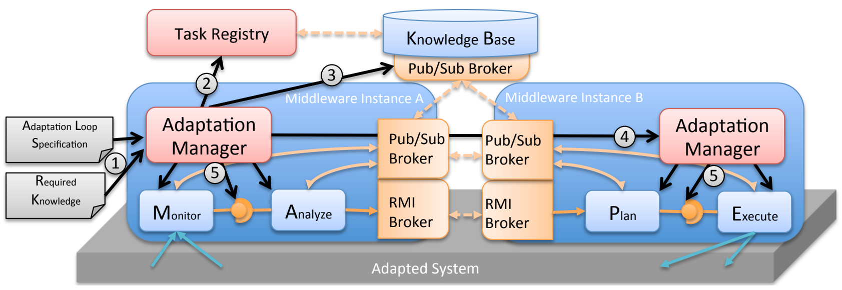


Figure 2: Construction of the adaptation engine

The developer decides to start with a predefined ALS which requires only simple action policies. Using this ALS, he can write, for instance, an action policy to adjust the media frame rate (action) delivered to the consumers according to the monitored network bandwidth availability (condition). After experimenting with action policies, the developer observe that the policies can not properly express trade-offs between the controlled properties. Trying to reach optimal operational states, he decides to exchange the policies with utility functions. With a rigid framework, this decision would require major modifications to the adaptation engine and in some cases the exchange of the middleware solution. With the proposed framework, the developer only needs to exchange the *analyzing* and *planning* tasks of the ALS, and redeploy it together with the new required knowledge (utility functions, quality model and quality predictors).

Considering that every user of the application is equally important and that they may be sharing resources (e.g. network bandwidth), the goal of the system now is to find a feasible set of adaptation actions that maximize the utility of the application for all users. A centralized planning solution may became non-scalable as the number of users grows, due to the time and resources taken to evaluate all possible adaptation actions and find the best. Aware of this limitation, the developer specify a utility function that describes the utility of the planning task as a function of the number of users, the time taken to find a new plan and its optimality (decentralized algorithms using only partial knowledge may find sub-optimal solutions). This utility function is deployed together with a new ALS containing a simple quality control task (as the one shown in Figure 1(b)). The new control task is dynamically created by the adaptation manager and now the adaptation engine itself is also subject to self-adaptation.

5 Related Works

In this section we refer to a few related projects and their limitations compared to our approach. In [MDL10], the authors propose a framework for developing systems of systems with autonomic capabilities. The framework supports different types of adaptation policies (action, goal and utility functions) and hierarchical cooperation between distributed managers. However, it does not provide a model with loosely coupled control task or consider evolvability in terms of the quality of the adaptation process. The framework proposed in [CK10], is based on opportunistic composition of loosely coupled service-oriented control tasks. The opportunistic nature can deal with unanticipated states but can also potentially lead to unwanted behavior. Furthermore, it implies little visibility to the adaptation loop and limits the interaction to an event based model.

6 Conclusions and Future Work

In this paper, we presented our vision of a flexible and evolvable framework for self-adaptation, where the self-adaptive behavior is built reusing different approaches for its constituent control tasks and where the adaptation engine can evolve over time. We intend to refine the OWL-S ontology used to describe the adaptation process and integrate our solution into an adaptation middleware to demonstrate its feasibility. We are considering existing middleware solutions with open architectures that can be easily modified and extended with a process execution engine, such as QuA [GEA06]. We expect to apply the final framework to explore customized, decentralized and evolvable approaches for self-adaptation to new large-scale application domains.

Bibliography

[BCNR10] D. Breitgand, R. Cohen, A. Nahir, D. Raz. On cost-aware monitoring for self-adaptive load sharing. *Selected Areas in Communications, IEEE Journal on* 28(1):70 –83, 2010.

[CK10] R. Calinescu, M. Kwiatkowska. Software engineering techniques for the development of systems of systems. *Foundations of Computer Software. Future Trends and Techniques for Development*, pp. 59–82, 2010.

[GEA06] E. Gjørven, F. Eliassen, J. Aagedal. Quality of Adaptation. In *2006 International Conference on Autonomic and Autonomous Systems, ICAS '06*. Pp. 9 –9. 16-18 2006.

[KC03] J. Kephart, D. Chess. The vision of autonomic computing. *IEEE Computer* 36(1):41 – 50, Jan. 2003.

[MBM^{+}07] D. Martin, M. Burstein, D. McDermott, S. McIlraith, M. Paolucci, K. Sycara, D. McGuinness, E. Sirin, N. Srinivasan. Bringing Semantics to Web Services with OWL-S. *World Wide Web* 10:243–277, 2007.

[MDL10] Y. Maurel, A. Diaconescu, P. Lalanda. CEYLON: A service-oriented framework for building autonomic managers. In *2010 Seventh IEEE International Conference and Workshops on Engineering of Autonomic and Autonomous Systems*. Pp. 3–11. 2010.

[MPS08] H. Müller, M. Pezzè, M. Shaw. Visibility of control in adaptive systems. In *Proceedings of the 2nd international workshop on Ultra-large-scale software-intensive systems*. Pp. 23–26. 2008.

[ST09] M. Salehie, L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 4(2):1–42, 2009.