Engineering Self-adaptive Systems: From Experiences with MUSA to a General Design Process

Massimo Cossentino[1], Luca Sabatucci[1(✉)], and Valeria Seidita[1,2]

[1] Consiglio Nazionale delle Ricerche, Istituto di Calcolo e Reti ad Alte Prestazioni, Palermo, Italy
{massimo.cossentino,luca.sabatucci}@icar.cnr.it
[2] Dip. dell'Innovazione Industriale e Digitale, Università degli Studi di Palermo, Palermo, Italy
valeria.seidita@unipa.it

Abstract. Designing and developing complex self-adaptive systems require design processes having specific features fitting and representing the complexity of these systems. Changing requirements, users' needs and dynamic environment have to be taken in consideration, also considering that, due of the self-adaptive nature of the system, the solution is not fixed at design time but it is a run-time outcome. Traditional design approach and life cycles are not suitable to design software systems where requirements continuously change at runtime.

A new design process paradigm is needed to design such systems. In this Chapter, we present a retrospective analysis based on three projects developed in the last five years with the middleware MUSA in order to identify specific features of the design process for supporting continuous change and self-adaptation. The result is a general approach allowing to reduce the gap between design time and run-time.

Keywords: Adaptive management · Continuous change · Design process

1 Introduction

Today, there are several trends that are forcing application architectures to evolve. Users expect a rich, interactive and dynamic user experience on a wide variety of clients including mobile devices. Customers expect frequent rollouts, even multiple times a day, to keep pace with their informational and service requirements. Moreover, customers want to significantly reduce technology costs and are unwilling to fund technology changes that do not result in direct customer benefits.

In traditional software life-cycles, a single change can affect multiple components, creating a complicated testing effort, requiring testers to understand various code interdependencies or test the entire application for each change. IT

© Springer Nature Switzerland AG 2019
D. Weyns et al. (Eds.): EMAS 2018 Workshops, LNAI 11375, pp. 96–116, 2019.
https://doi.org/10.1007/978-3-030-25693-7_6

organizations demand a paradigm shift: from monolithic applications (that puts all user interfaces, business logic and data in a single process) toward applications that enable architectural extensibility.

The level of adaptability to changing requirements is managed at design time using ad-hoc life cycle or process models. Developing self-adaptive systems using a systematic approach requires to consider several factors that may be summarized in: changing operational context and changing environment.

Even though different kinds of approaches for engineering self-adaptive systems exist - they span from control theory to service-oriented and from agent-based approaches to nature-inspired ones - today some possible good approaches seem to be those exploiting models-at-run-time and reflection. Nevertheless, a disciplined and systematic design process for developing self-adaptive systems, able to consider changing operational context and changing environment, still lacks.

A need for new design paradigms arises. Design paradigms where continuous changes are managed through continuous delivery or adaptation of new portions of the system during its operations.

The aim of this Chapter is to identify a general design process for self-adaptive systems. For pursuing this objective, we started from our experience with MUSA (Middleware for User-Driven Service Adaptation), the middleware we created for developing self-adaptive systems. We explored the way in which MUSA works, to identify and to analyze which are the elements of the process involved in that. We considered a five years experience in employing MUSA on three projects. The analysis has been conducted focusing on the design activities and three different measures to gain insights on the effort spent in the design and the aptitude of the system to autonomously find solutions. The results have, then, been used for generalizing a design approach.

The rest of the Chapter is organized as follows: Section 2 discusses the need for a new design paradigm, Sect. 3 illustrates some existing middleware for self-adaptive systems, Sect. 4 discusses the retrospective analysis on MUSA; in Sect. 5 the obtained results are discussed and in Sect. 6 we discuss them and we propose a general design approach; finally in Sect. 7 some conclusions are drawn.

2 Continuous Changes and Self-adaptation

A self-adaptive system is a system able to modify its behavior and/or its structure in order to respond to changes perceived from the environment it is working on or from inside the system itself. Changes are considered to occur while the system is working. System requirements and also system ability to adapt depend by all the actors that interact with the system, the environment whose changes are affected by and affect the system. The system behavior itself is a source of changes and adaptation. Adaptive behavior is prone to three types of dependency: actor-dependency, system-dependency, and environment-dependency.

Designing and developing self-adaptive systems have to consider the following factors: requirements are identified at runtime, environment conditions continuously change, users heavily and continuously interact with the system and the global behavior of the system emerges at runtime.

Traditional software engineering approaches cannot be used for developing self-adaptive systems. They prescribe a very disciplined process that follows a well-specific life-cycle; the main aim is to make the software development as more predictable as possible. All the requirements have to be identified and analyzed in the very early activities of the design process and then transformed into code.

A software system is a solution to a problem, regardless the level of complexity of the problem and the software, and the level of adaptivity to changing requirements is managed at design time using ad-hoc life cycles or process models. Several process models may be used: waterfall, iterative and incremental and so on. This way of working has been well established since years for all those systems that do not require particular changes and do not work in changing conditions.

Different kinds of approaches for engineering self-adaptive systems exist [14,22,41], they span from control theory [10] to service-oriented [29] and from agent-based approaches [19,40] to nature-inspired ones [43]. They all map to the so-called MAPE cycle [38]: monitoring, analyzing, planning and executing. In [42], for instance, authors propose five patterns of interacting MAPE loops to be used for implementing decentralized control.

A promising approach to manage complexity in runtime environments is to develop adaptation mechanisms that involve software models. This is referred to as models@run.time. The idea is to extend the model produced using MDE approaches to the run-time environment. The authors of models@run.time emphasize the importance that software models (artifacts) may play at runtime stating that *if a system changes the representation of the system should change and vice versa.* Another approach aims at managing complexity in runtime environments and at implementing MAPE cycle by developing adaptation mechanisms that involve software models (artifacts). Blair et al. [6] emphasize the importance that software models may play at runtime. They use the mechanism of reflection inducing that the necessary adaptation is performed at the model level rather than at the system level.

This vision is opposite to the traditional design approaches that prescribe the system be stopped each time a new requirement or a new goal occurs or the environmental conditions change. Conventional processes also prescribe that changes have to be inserted in the system while it is not working; there are several methods in literature for facing changes, from simple software maintenance to software evolution [39]. In any case, a new design activity is necessary during which the system cannot work; someone says that in this case, the system is offline.

In our work, we consider that designing and developing self-adaptive systems require continuous delivery and designing while the system is working; we accept the idea the system has to be always online.

Baresi et al. [3] introduce the need of bringing near the design time to the runtime: "The clear separation between development-time and run-time is blurring and may disappear in the future for relevant classes of applications". This allows some changing activities to be shifted from design and development to runtime

and some changing responsibilities to be assigned to the system itself instead of to the analysts or designers. Thus, realizing and implementing adaptation [1,9].

Several scientists agree [2,13] self-adaptation is closely related to the ability of reasoning about the inner world beyond than the outer. In other words, self-awareness is the key for self-adaptation. In previous work, we adopted agent's knowledge to implement run-time artifacts for modeling user's requirements and norms [32].

We claim that the maintenance of complex distributed software is a mix of continuous delivery and continuous integration. Automation is indeed one way to enable constant changes. In particular, we are interested in exploring the automation that supports continuous changes. Hence, continuous changes may be handled at runtime with the aid of an automatic tool.

Our intuition is that the life cycle of a self-adaptive system, or of one of its components, starts with its design and does not terminate with its deployment [25] and testing. The life-cycle continues with some monitoring phases aiming at identifying and handling new or emergent requirements and/or needs from users. This implies that self-adaptation allows making run-time changing and the self-adaptive system itself supports the further development phase aiding, or better substituting, designers. In so doing, we overcome the limits of traditional design methodologies: they are not adequate for a self-adaptive system because they do not consider the run-time.

In the following sections, we identify the characteristics of a design process for supporting self-adaptive middleware. To investigate this topic we exploit the experience gained with MUSA, a self-adaptive middleware, and then we generalize some of the obtained results.

3 MUSA: A Middleware for Self-adaptation

MUSA (Middleware for User-driven Service Adaptation) [34] is a middleware for orchestrating distributed services according to unanticipated and dynamic user needs. It has been conceived for managing evolution and adaptivity of dynamic workflows [33]. MUSA provides basic concepts to model a software system able to detect and react to exceptional events, failures and resources unavailability.

Key enablers of MUSA are: (a) representing *what the system will do* and *how the system can do* as a couple of first-class entities (respectively *Goals* and *Capabilities*) [36]; (b) providing goals and capabilities and run-time artifacts the system can reason on by representing them through a common formalism, based on a grounding semantic [17]; (c) providing a flexible and configurable planning system [31] for dynamically generating workflows of capabilities to address the specified goals.

In the following, we compared MUSA with some of the middleware for self-adaptation in the literature. The remaining sub-section will discuss which steps are necessary for engineering a system with MUSA.

3.1 Middlewares for Self-adaptation

Literature provides an increasing number of middlewares for developing and managing the self-adaptive characteristics of a system under development. These approaches are highly heterogeneous, yet one can usually classify them as component or service-based [4,29], agent-based [14,31], or bio-inspired [20,43].

The benefits of these middlewares are that they provide basic functionalities for rapid prototyping of many self-adaptation features such as monitors and actuators. The common factor of almost all these different infrastructures is the idea of exploiting the mechanism of reflection to take run-time strategic decisions. They support some run-time entities and models, i.e., high-level abstractions of the software system. By maintaining these abstractions at run-time, the software system could be able to perform reflection, and it may predict/control certain aspects of its behavior for the future.

Kramer and Magee [8,24] propose MORPH, a reference architecture for self-adaptation, inspired to robotics, that includes (i) a control layer, a reactive component consisting of sensors, actuators and control loops, (ii) a sequencing layer which reacts to changes from the lower levels by modifying plans to handle the new situation and (iii) a deliberation layer that consists in time consuming planning which attempts to produce a plan to achieve a goal. The main difference with our architecture is that we introduce a layer for handling goals evolution. The architecture is suitable for implementing a self-adaptive system able to deal with anticipated changes by selecting among pre-computed adaptation strategies.

SeSaMe (SEmantic and Self-Adaptive MiddlewarE) [5] is a self-organizing distributed middleware that uses semantic technologies to harmonize the interaction of heterogeneous components. In SeSaMe, components self-connect at run-time, without any prior knowledge of the topology. The dynamic architecture grants system's reliability even when multiple components leave or fail unexpectedly, and dynamically alters the system's topology to cope with message congestion. The main difference is that SeSaMe focuses on structural/component-based abstractions (groups, roles, components) whereas MUSA concentrates more on functional requirements. In SeSaMe, adaptation consists in modifying the topology of connections among the components, whereas in MUSA it consists in changing the workflow by removing/replacing infeasible tasks.

SAPERE (Self-aware Pervasive Service Ecosystems) [43] is inspired to natural ecosystems to model dynamism and decentralization in pervasive networks. In SAPERE various agents coordinate through spatially-situated and environment-mediated interactions, to serve their own needs as well as the sustainability of the overall ecology. The environment is modeled as a spatial substrate where agents' interactions are managed as virtual chemical reactions. The main difference is that SAPERE is focused more on emergence and evolution rather than on control. The collaboration between agents is incidentally due to the current context and to underlying eco-laws. Emergence is programmed via eco-laws, i.e. natural metaphors that specify how agents will interact.

As a final remark, independently from the kind of middleware chosen for a specific purpose, all of them imply a methodological shift in which some design models move from the design-time towards run-time artifacts.

3.2 Using MUSA for Engineering a Self-adaptive System

The use of MUSA for building a self-adaptation system consists basically in providing a model of goals and a model of capabilities to the MUSA instance, thus enabling its proactive means-end reasoning. However, there exist a preliminary activity to be done: the analysis of the domain for building a common ontological background for goals and capabilities.

The *Problem Ontology Description* (POD) is a design fragment [28] that allows describing the problem domain elements and their relationships in a formal way. This activity grounds on an ontology used as an analysis (i.e. descriptive) model for representing the reality of problem domains typically addressed by agent-oriented technologies. This ontology is described by the Problem Ontology metamodel. The Problem Ontology metamodel, we employ, has been inspired by the FIPA (Foundation for Intelligent Physical Agents)[1] standard and ASPECS [16] ontology. Thus, similarly, our meta-ontology describes what are the elements of interest in a domain (*Concept*) with their properties (*Predicate*) and how they act in the domain (*Action*) and it introduces some new elements in order to explicitly model intentional behaviors.

Requirement Analysis and Goals. Traditionally, when specifying system requirements, analysts crop the solution space in order to define the expected system behavior in a deterministic way. However, the characteristics of being autonomous and proactive make the agents able to explore a wider solution space, even when this space dynamically changes or contains uncertainty [40]. The novelty of our approach consists in making some constraints of the solution less rigid, thus allowing more degrees of freedom to the system.

Several methods exist in literature to conduct a goal-oriented requirement analysis. We do not suggest to use a specific one, providing the output is rendered via the GoalSPEC language [37]. It has been specifically conceived to support MUSA with a run-time artifact for dealing with user's requirements, some of its most interesting features will be presented in the following.

GoalSPEC Supports Adaptivity. GoalSPEC provides some domain-independent keywords but it offers a powerful plug-in mechanism for providing different ontology groundings. It is fully compatible with the Problem Ontology Description fragment, thus goals can be expressed as desired states of the world, defined in the POD as concepts and predicates.

GoalSPEC Supports Evolution. GoalSPEC allows MUSA agents to reason and commits to the specified goals. Goals are run-time artifacts, therefore agents perceive them as part of the environment. This run-time nature of goals allows they can change during system lifecycle, thus supporting a global evolution of the system.

[1] Available at: http://www.fipa.org/specs/fipa00086.

Services and Capabilities. The concept of capability comes from AI (planning actions [21]), software engineering (contracts [18]) and service-oriented architecture (micro-services [26]). Indeed, this composite nature is well represented by the separation we adopt between *abstract capability* – a description of the effect of an action that can be performed – and *concrete capability* – a small, independent, composable unit of computation that produces some concrete service.

Implementing system functionalities as capabilities provides some benefits:

– each capability is relatively small, and therefore easier for a developer to implement,
– it can be deployed independently from other capabilities,
– it is easier to organize the overall development effort around multiple teams,
– it supports self-adaptation because of improved fault isolation.

An example of description of capabilities is provided in [34] where the smart travel domain is considered. In this context, each capability encapsulates a web service for reserving some kind of travel service (hotel, flight, local events).

Moreover, we focused on the idea that capabilities make it easier to deploy new versions of the software frequently. Providing capabilities (as well as goals) as run-time entities contributes to enable continuous changes and self-adaptation. Supporting this claim is one of the objectives of this Chapter. In the remaining section, we used data about the implementation of three different applications for getting some findings of the easiness of continuously evolving a system.

4 A Retrospective Analysis of MUSA

This section presents a retrospective analysis of the design activities with MUSA and discusses some emerging results.

Empirical Study Design. We selected MUSA [34] because we gained a practical experience of use, due to its adoption in several applications.

In the last years, MUSA has been employed in research projects and case studies with very different application domains. Table 1 gives an overview of the sources from where data have been collected.

The empirical study mainly focuses on the design activities for producing ontology, capabilities and goals for the selected projects of Table 1. The design process we followed in all the projects follows three main activities:

– As it happens in traditional requirement analysis, we suggest every MUSA project started with a good understanding of the domain. We adopt an ontology to record and represent this knowledge. For this reason measuring the evolution of the **ontology** model may be interesting for this study.

Table 1. Summary of research projects and case studies where the MUSA middleware has been employed between 2013 and 2016.

Acronym	Type	App. Name	Description
IDS	Research Project	Innovative Document Sharing	The aim has been to realize a prototype of a new generation of a digital document solution that overcomes current operating limits of the common market solutions. MUSA has been adopted for managing and balancing human operations for enacting a digital document solution in a SME
OCCP	Research Project	Open Cloud Computing Platform	The aim was the study, design, construction and testing of a prototype of cloud infrastructure for delivering services on public and private cloud. MUSA has been employed, in the demonstrator, in order to implement an adaptive B2B back-end service for a fashion company
Smart Travel	Case Study	Travel Agency System	MUSA provides the planning engine that creates a travel-pack as the composition of several heterogeneous travel services. The planning activity is driven by traveler's goals

- The second step is understanding and representing customer's requirements. In MUSA, they must be translated into significant states of the system to be addressed. In some circumstances, this activity may require a revision of the ontology to adjust some of the concepts. For this reason, the study includes an evaluation of the evolution of the **goal** model.
- A third step concerns the development of the services the system may employ in the emerging solution. In MUSA, capabilities are run-time artifacts that describe how to employ available services to compose a solution. As well as goal modeling, defining the capabilities may require a revision of the ontology. Therefore, we decided to include the analysis of the available **capabilities**.

For comparison reasons, for each project, we identified three main iterations, in which the application received substantial changes. In different projects, iterations have been deduced by considering the delivery of functionalities, therefore they may have a variable duration between 1 to 2 months. In each iteration, we have considered either which artifacts have been produced or how they have been modified with respect to the previous iteration (versioning history).

For each artifact, we planned a set of measurements.

- The first measurement is the **size** of the model. It is calculated by employing the system metamodel as illustrated in [7]. The metamodel provides the language for describing models of the system. It contains elements and relationships underpinning and guiding the design process activities used for developing a specific system. During the design activities, designers use the metamodel as a trace for instantiating elements in the models. The **size** of the model is a measure of the effort spent on instantiating models from the metamodel. It refers to introducing new elements, relationships, attributes and so on.
- The second measurement is the **effort** (in man-hours) spent in the model. This measure is calculated by considering the number of commits done for

the specific artifact. To be more precise, we asked the involved developers to confirm or adjust the values. In any case we considered a possible error in this measure, thus we considered significant the differences of effort rather than their absolute values.
- The last measurement is done on the running system. After injecting the new set of goals and capabilities (by replacing the previous ones), MUSA calculates a new space of configurations and extracts a number of solutions to be used to provide the requested functionality. The measurement is done on the space of configuration as a value of the degree of freedom of the adaptation mechanism. It provides two values: the **number of** different **solutions** computed by the system for solving the problem.

5 Interpretation of Results

Table 2 reports the empirical data extracted from the three projects during their initial three iterations. Data is also summarized in three charts, as shown in Fig. 1.

The use of MUSA implies, at the very beginning, to perform some classical design activities. After the first injection, the self-adaptive application is online and every required change may be handled while it is running. We use the empirical data for identifying duration/effort of the various release phases of the process necessary for delivery a self-adaptive application with MUSA.

Before examining data, it is useful to provide some additional details about how MUSA works. MUSA is based on the paradigm of collaborating agents and artifacts [27]. Figure 2 depicts the main stakeholders, agents and artifacts involved in this process. According to the classic vision, an agent can perceive the environment and act in order to change it. In addition, MUSA agents are self-aware of which capabilities they own and how to use them for producing a result. MUSA agents share a main goal: ‘to address users’ run-time goals’ (i.e. requirements). Therefore they continuously monitor either goal injection or goal changes.

When the designer specifies a set of goals to be addressed (or update them), then the agent groups called *solution explorer* is ready to collaborate to find one or more abstract solutions (as workflow of abstract capabilities). These form a run-time model called *Solutions* artifact (Fig. 4). The algorithm is described in [31,35].

Now, we use data from Table 2 to specify how these participants (humans and agents) collaborate during design-time and run-time (Fig. 4). It is worth noting that we should address two different system layers in studying MUSA applications: the MUSA middleware and the MUSA application:

1. The MUSA middleware provides runtime facilities for goal-models and capabilities, and enables agents for solution-discovery and adaptive-orchestration.
2. The MUSA (self-adaptive) application is the result of employing the MUSA middleware in building a set of user’s requested functionalities. It is able to adapt to a changing domain.

Table 2. Summary of the empirical data by retrospective analysis of research projects in which the MUSA middleware has been adopted for engineering a self-adaptive system

Project	iteration 1	iteration 2	iteration 3
IDS	first injection	bugfix+evolution	evolution
size (number of model elements)			
ontology	6	9	10
capability	4	6	7
goal	4	6	7
effort (man hours)			
ontology	10	7	3
capability	30	23	7
goal	14	7	1
design total effort	54	35	21
space of configuration (number of solutions)			
	1	6	6
OCCP	first injection	evolution	bugfix+evolution
size (number of model elements)			
ontology	10	10	10
capability	5	8	12
goal	8	8	9
effort (man hours)			
ontology	30	10	7
capability	70	40	50
goal	7	7	4
design total effort	107	57	61
space of configuration (number of solutions)			
	1	9	18
Smart Travel	first injection	bugfix+evolution	bugfix+evolution
size (number of model elements)			
ontology	12	12	14
capability	3	5	8
goal	5	7	7
effort (man hours)			
ontology	7	7	14
capability	40	25	20
goal	100	14	1
design total effort	147	46	35
space of configuration (number of solutions)			
	5	5	5

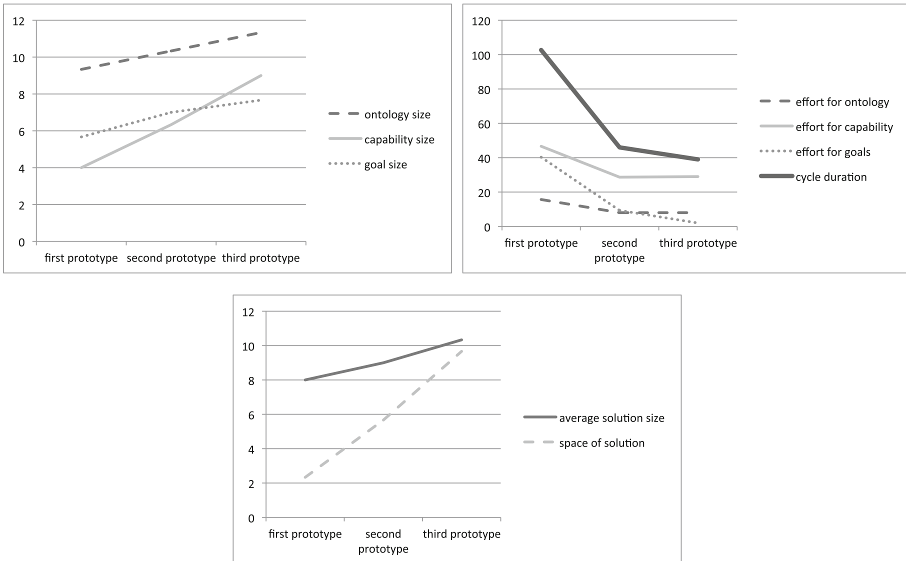

Fig. 1. Charts reporting average data, along three iterations, as extracted for the three projects. Top-left diagram shows the average increase of the complexity of the ontology, the capability model and the goal model. Top-right diagram shows the corresponding effort (in man hours) required to complete the iteration. Finally, bottom diagram highlights the growth of the space of solutions.

Design-time and run-time phases are represented in Fig. 3 in which we highlight the MUSA middleware and the MUSA (self-adaptive) application areas.

Design-time, generally speaking, is the moment in which taking design choices concerning the characteristics of the application. The *design-time of the MUSA Middleware* is out the scope of this Chapter. We work under the hypotesys that MUSA is complete and always running.

Therefore, Fig. 3 focuses on MUSA Middleware run-time phase (right-side of the box on the bottom). The box on the top represents the two phases of the MUSA application: design-time and run-time.

The *design-time for MUSA applications* concerns the definition of an ontology and, subsequently, of a couple of artifacts: goals and capabilities. As shown in Fig. 3, this design-time of the MUSA application occurs during MUSA middleware run-time, indeed the designer may exploit some simulation facilities offered by the middleware for evaluating the degree of adaptation of the application as a consequence of the new specifications.

The *run-time of the MUSA application* is shown in the top-right box of Fig. 3. It includes two possible states: online and offline. Offline is when the application is executing background operations but it not provides a working response to user expectations; on the other side, online means the application is providing the expected functionalities. The application is offline before the

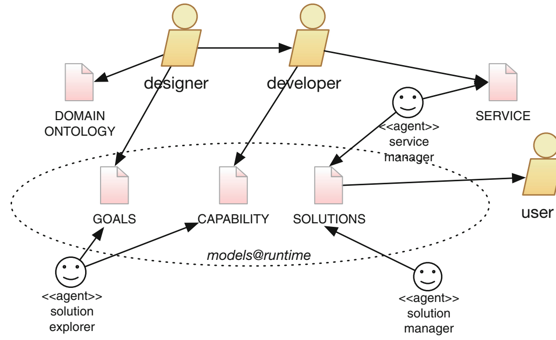

Fig. 2. The Human-Agent collaboration for the development of a MUSA self-adaptive application.

first injection (of goals and capabilities) and after any future injections for bug-fixing or functionality evolution: during this phase the middleware layer provides functionalities for solution-discovery. When the application layer is online, the middleware operates with an adaptive orchestration.

It is worth noting that, in the fashion of a continuous software delivery, the red line of Fig. 3, i.e. the boundary between MUSA application design-time and run-time, is less clear than the blue one. Indeed, after the first injection, designing the MUSA application may be an activity performed during application is online. Clearly, when changing the specifications at runtime, a short interruption of service occurs due to the adaptation activity.

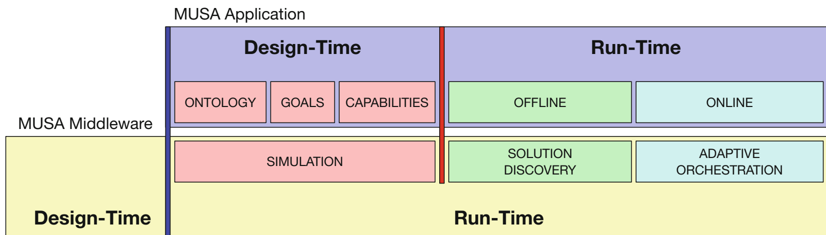

Fig. 3. States of MUSA run-time execution. (Color figure online)

In Fig. 4, the alignment represents the design activities corresponding to a particular artifact (ontology, goals, capabilities and architecture). Indeed, MUSA agents support designer and developer, respectively, (1) by evaluating the degree of freedom of the set of goals and capabilities that are going to be built, and, (2) by verifying the compliance of the service under development with the corresponding capability. Figure 4 highlights this collaboration by coupling humans and agents in a design activity (designer with solution explorer and developer with service manager).

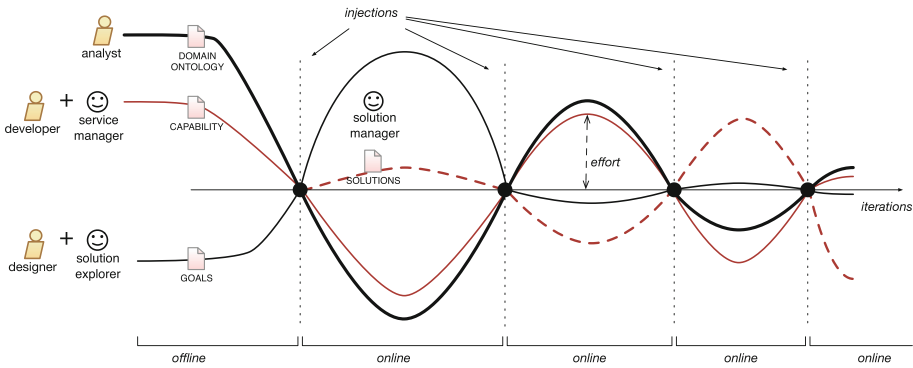

Fig. 4. Outcome of the retrospective analysis.

According to previously reported data, we identified three iterations (or release cycles) that begin and terminate with an injection. In Fig. 4, the amplitude of lines are proportional to the effort required for refining the correspondent artefact.

Results of the retrospective analysis are summarized in Fig. 4 and resumed in the following findings.

Injections. The boundary between the offline and online design is marked by the first injection, that is the moment in which the self-adaptive application begins. Time before the first modification point, the left part of the figure, represents when analyst, developer and customer designed the first version of the system for solving a specific problem with the aid of some agents working in MUSA. The short time interval soon after an injection is used by MUSA for acquiring occurring changes in the operating condition and for releasing new configurations of the system.

Boundary between MUSA Application Design-Time and Run-Time. The developer designs a first set of capabilities the system has to own and the designer designs a first set of goals the system has to pursue by using the right capabilities. These two design-time activities are performed respectively, with the aid of the service manager agent and the solution explorer agent. In particular, the solution explorer aids the designer in evaluating if available capabilities are enough for addressing the set of goals, also indicating the degree of freedom for future adaptations. The classical boundary between design-time and run-time is going to disappear.

Solution as a Model at Run-Time. Once the self-adaptive application is online, agents collaborate in order to achieve the goals and to monitor the environment. They exploit the available capabilities of the selected solution. The solution is a run-time artifact that only agents are responsible for (no human role is involved). They may change it for adaptation purpose.

Convergence. After each modification point, each design iteration takes a short time and less effort to be completed; on the other hand, the space of solutions increases. We observed that the self-adaptation property contributes in reducing the design effort. This because, iteration after iteration, the ontology domain description becomes stable and the repository of capability increases. As a consequence, the self-adaptive application is able to endorse a higher number of deviations from standard situations. Every time a modification occurs in the running/operating conditions (for instance, a new goal, a change in the environment or a change in the way the user uses the system), it is less frequent designers start a new design iteration. However, when a manual change is required, the ontology allows to quickly refine goals and to specify new capabilities. In practice, in the long run, designers and agents will interact less and for short time.

6 Discussion

In this section, starting from the results of the retrospective analysis, we propose a skeleton of a design process for engineering self-adaptive systems. We will achieve this objective by extracting a schematic design process from our previous experience with MUSA and then trying to generalize it.

6.1 The Design Process Adopted in MUSA Applications

It is a matter of fact that exploring the new world of adaptive system has brought many research groups to move in a new context where old methodologies have soon proved to be not applicable. Similarly, when starting our experiences with MUSA we tried to employ design activities and related artifacts coming from our agent-oriented software engineering background. Notably we considered influences coming from PASSI [15], Agile PASSI [12], and ASPECS [16]. Some of them influenced not only our way to use MUSA but, as it was expectable, the development of MUSA itself. For this reason, we will find some of them in the process we are trying to sketch as a suggested approach to the design of self-adaptive systems. Another relevant issue to be considered when looking at the way we designed our MUSA-based solutions is that MUSA itself was quickly and drastically evolving. Mostly in the first part of this 5-years long observation period. The fundamental concepts MUSA is based on (goals, capabilities, agents' hierarchical organization and so on) remained unchanged but their contribution to the middleware implementation significantly evolved over time. Looking at how we effectively developed the solutions required in the different projects where we employed MUSA, we can see the constant presence of the following design activities:

- Ontology definition
- Goals definition (and injection)
- Capability definition (and injection)

– Problem Solution
– Adaptation loop

These activities will be detailed in what follows.

Ontology Definition. One of the key ideas at the basis of MUSA is to let different modules contribute to find a solution to a problem even if they have not been conceived for that. In order to do that all the system parts (at least those involved in the solution of a specific problem, others may exist that are not involved on this but will contribute to another) need a common semantics for sharing information. Following the influences coming from our past experiences in software design we decided to provide that by using a methodology. Such methodology not only describes the concepts in the solution domain (and their status by using predicates) but also actions allowed in the domain itself.

Goals Definition (and Injection). In order to employ MUSA for solving a problem the designer has to communicate the problem requirements to MUSA. This is done by using goals and more specifically by injecting them in the (already) running middleware. Such goals will be received by a Solution Explorer agent who will be in charge of pursuing them as already depicted in Figs. 2 and 4. Goals will be expressed by referring to the problem ontology produced by the previous activity.

Capability Definition (and Injection). In our projects, capabilities often come from the real world. For instance, existing web or cloud services. A great part of the capabilities construction effort therefore consisted in wrapping them in order to ensure a semantically effective interaction with MUSA.

Problem Solution. This is the moment when MUSA is asked to solve the problem. MUSA uses its reasoning algorithms in order to find an abstract solution (employing abstract capabilities) and if feasible binds that to executables modules/services (concrete capabilities).

Adaptation Loop. There are several reasons that may trigger this loop: the execution of a concrete capability may not reach the expected result, the module/service wrapped by the capability is no more working or the proactive means-end reasoning module does not find a solution to the injected set of goals. MUSA reacts to such situations in two different ways: firstly, it tries to overcome the obstacle by replanning the solution (at the concrete or abstract level), finally, if the other ways did not solve the issue, MUSA involves the user in the loop by asking for its collaboration in terms of goals changing, constraints relaxing or injection of new capabilities.

According to our experiences these activities are all crucial and constantly applied in the design of our systems. Because of that we think these activities may be the pillars for building a more general design process as it will be discussed in the next subsection

6.2 A Generalised Design Process

A crucial part for any design process is to define its application scope. We think our experience is representative for the following category of systems and problems. First, the system is composed of two layers:

1. A middleware layer providing assembling/orchestrating/coordinating features of existing pieces of functionalities (software) and providing adaptation features to cope with unforeseeable changes.
2. An application layer running upon the middleware one. This layer directly interacts with the user providing the required functionalities/solutions.

Elementary pieces of functionalities assembled in the solution may be described in a semantically coherent way. The problem may be described in terms of functional and non-functional requirements that may change during system execution and that may be expressed in a machine understandable language. The problem requires the system to adapt to unforeseeable changes in the environment and in the system itself (requirements included) so that it can find solutions that may employ different strategies/portions of software/parameter settings.

For such a category of problems/systems we think the following process skeleton may be successfully applied.

1. Define problem and solution taxonomy or other semantic description. This creates an operational abstraction where the problem may be consistently described in terms of requirements and solution elements (composition/employment of existing pieces of software, data types, etc.).
2. Define problem model in a machine-readable language. Problem requirements cannot be expressed in conventional design languages (for instance UML) since the application-layer of the system has to be aware of them, both at the functional and non-functional level of detail.
3. Collect and wrap existing functionalities. The result will be a repository of semantically interoperable pieces of software that the adaptation middleware may compose to obtain the solution.
4. Validate functionalities repository towards requirements satisfaction. To this purpose, some relevant works on certification of self-adaptive systems may be found in literature [11,30]. Another challenge for adaptive systems is to ensure that enough pieces of functionality are available to face the demands of change proposed by the environment, changing user needs, system failures and so on. This check is a relevant issue and we think each middleware should interact with the designers (for instance using simulation features) in order to verify if the existing repository ensures a sufficient degree of adaptation. The specific algorithms used by the middleware may deeply affect the results of this validation.
5. Run the application layer. This may be roughly compared to the conventional running phase of a traditional software. The application layer needs the models produced in the previous phase in order to learn and pursue the specified

objectives. If the solution is not found or when it fails after succeeding for a while, the next activity will follow.
6. Adaptation. According to different implementation philosophies (and problem constraints) this phase may involve the human or not. For instance, when sensitive decisions have to be taken, a human supervision is usually required before swerving from a straightforward solution. Adaptation may involve the employment of alternative pieces of functionalities in pursuing the same plan, a replanning of the solution strategy or other approaches (for instance evolutionary ones) according to the specific middleware.

6.3 Limits of This Analysis

The data extracted could be a bit biased because in these three projects, engineers developed both the MUSA middleware and the MUSA application for the specific domain problem. In the reported retrospective analysis the most complex part was separating the time required for fixing the middleware from the time required to implement the application (ontology, goals and capabilities).

Moreover, we have restricted the retrospective analysis to the first three iterations. However, some projects were developed in more iterations that were not considered in this analysis. This choice was done in order to make them comparable. In any case, Fig. 1 shows that the trend of the curves is quite regular. So we can hypothesize the sample is quite respectful of the reality.

7 Conclusions

Due to the features of self-adaptive systems and the fact that, nowadays, systems are more interconnected and various than before, designers have not the right means to anticipate and design interactions among different components, interaction among users and the system. Indeed, (self-adaptive) software system properties are effectively known when all the relationships among the software components and between the software and the environment have been expressed and have been made explicit. Such issues have to be dealt with at runtime; modeling and monitoring users and the environment is the key for enabling software to be adaptive [13,23].

Self-adaptation deals with requirements that vary at run-time. Therefore it is crucial that requirements lend themselves to be dynamically observed, i.e., during execution. Middlewares for self-adaptation constitute the right tools for easing complex systems development and for providing a form of model@runtime. A methodological approach for developing self-adaptive systems supporting runtime continuous change still lacks.

In this Chapter, we illustrated the results of a retrospective analysis conducted on our middleware (MUSA) to identify the characteristics of a design process for developing self-adaptive systems.

We started from the hypothesis that changes occurring at run-time have to be handled by the system itself; like it were part of the team of designers. We reached this objective in MUSA by employing a well-specific agent architecture.

The analysis mainly highlighted that, using MUSA, supporting self-adaptive solutions implies a design process where humans and agents collaborate. Goals and capabilities are run-time entities that constitute the continuous data exchange between human and agents. Human and agents collaborate until the system (all the agents) possesses the useful knowledge for reaching the defined objectives by its own. Moreover, the collaboration between humans and agents and the fact that a run-time model exists until the system is running, guarantee the required system behaviour modifications during subsequent releases.

Finally, we deeply analyzed the way in which a system is developed by using MUSA and we identified some principal design activities a design approach for engineering self-adaptive system has to contain. The analysis of the use of MUSA covered five years. One of the most important insights we realized, also comparing that with other self-adaptive middleware systems, is that activities devoted to identifying the ontology of the system, the goals, and the capabilities are necessary to build a tool providing the right automation for supporting continuous changes.

The most relevant result of this analysis is the identification of the design process we used in developing MUSA applications. This process supports continuous change and strongly induce human and agents to collaborate in pursuing the solution. From this process we generalised a wider scope process for the design of self-adaptive applications based on the employment of a middleware layer providing assembling/orchestrating/coordinating features of existing pieces of functionalities (software) and providing the required adaptation features to cope with unforeseeable changes.

References

1. Andersson, J., et al.: Software engineering processes for self-adaptive systems. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems II. LNCS, vol. 7475, pp. 51–75. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35813-5_3
2. Aßmann, U., Götz, S., Jézéquel, J.-M., Morin, B., Trapp, M.: A reference architecture and roadmap for Models@run.time systems. In: Bencomo, N., France, R., Cheng, B.H.C., Aßmann, U. (eds.) Models@run.time. LNCS, vol. 8378, pp. 1–18. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08915-7_1
3. Baresi, L., Ghezzi, C.: The disappearing boundary between development-time and run-time. In: Proceedings of the FSE/SDP workshop on Future of software engineering research, pp. 17–22. ACM (2010)
4. Baresi, L., Guinea, S.: A3: self-adaptation capabilities through groups and coordination. In: Proceedings of the 4th India Software Engineering Conference, pp. 11–20. ACM (2011)
5. Baresi, L., Guinea, S., Shahzada, A.: SeSaMe: towards a semantic self adaptive middleware for smart spaces. In: Cossentino, M., El Fallah Seghrouchni, A., Winikoff, M. (eds.) EMAS 2013. LNCS (LNAI), vol. 8245, pp. 1–18. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-45343-4_1
6. Blair, G., Bencomo, N., France, R.B.: Models@ run.time. Computer **42**(10), 22–27 (2009)

7. Bonjean, N., Gleizes, M.-P., Chella, A., Migeon, F., Cossentino, M., Seidita, V.: Metamodel-based metrics for agent-oriented methodologies. In: Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 2, pp. 1065–1072. International Foundation for Autonomous Agents and Multiagent Systems (2012)
8. Braberman, V., D'Ippolito, N., Kramer, J., Sykes, D., Uchitel, S.: Morph: a reference architecture for configuration and behaviour self-adaptation. In: Proceedings of the 1st International Workshop on Control Theory for Software Engineering, pp. 9–16. ACM (2015)
9. Buckley, J., Mens, T., Zenger, M., Rashid, A., Kniesel, G.: Towards a taxonomy of software change. J. Softw. Maint. Evol. Res. Pract. **17**(5), 309–332 (2005)
10. Calinescu, R., Gerasimou, S., Banks, A.: Self-adaptive software with decentralised control loops. In: Egyed, A., Schaefer, I. (eds.) FASE 2015. LNCS, vol. 9033, pp. 235–251. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46675-9_16
11. Calinescu, R., Weyns, D., Gerasimou, S., Iftikhar, M.U., Habli, I., Kelly, T.: Engineering trustworthy self-adaptive software with dynamic assurance cases. IEEE Trans. Softw. Eng. **44**(11), 1039–1069 (2018)
12. Chella, A., Cossentino, M., Sabatucci, L., Seidita, V.: Agile passi: an agile process for designing agents. Int. J. Comput. Syst. Sci. Eng. **21**(2), 133–144 (2006)
13. Cheng, B.H.C., et al.: Software engineering for self-adaptive systems: a research roadmap. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 1–26. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02161-9_1
14. Cheng, S.-W.: Rainbow: cost-effective software architecture-based self-adaptation. ProQuest (2008)
15. Cossentino, M.: From requirements to code with the passi methodology. Agent-Oriented Methodol. **3690**, 79–106 (2005)
16. Cossentino, M., Gaud, N., Hilaire, V., Galland, S., Koukam, A.: ASPECS: an agent-oriented software process for engineering complex systems. Auton. Agents Multi-Agent Syst. **20**(2), 260–304 (2010)
17. Cossentino, M., Sabatucci, L., Seidita, V.: Towards an approach for engineering complex systems: agents and agility. In: Proceedings of the 18th Workshop on "From Objects to Agents", 1867, pp. 1–6 (2017)
18. Curbera, F.: Component contracts in service-oriented architectures. Computer **40**(11), 74–80 (2007)
19. De La Iglesia, D.G., Calderón, J.F., Weyns, D., Milrad, M., Nussbaum, M.: A self-adaptive multi-agent system approach for collaborative mobile learning. IEEE Trans. Learn. Technol. **8**(2), 158–172 (2015)
20. Fernandez-Marquez, J.L., Serugendo, G.D.M., Montagna, S.: BIO-CORE: bio-inspired self-organising mechanisms core. In: Hart, E., Timmis, J., Mitchell, P., Nakamo, T., Dabiri, F. (eds.) BIONETICS 2011. LNICST, vol. 103, pp. 59–72. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32711-7_5
21. Gelfond, M., Lifschitz, V.: Action languages. Comput. Inf. Sci. **3**(16), 1–41 (1998)
22. Haesevoets, R., Weyns, D., Holvoet, T.: Architecture-centric support for adaptive service collaborations. ACM Trans. Softw. Eng. Methodol. (TOSEM) **23**(1), 2 (2014)
23. Inverardi, P.: Software of the future is the future of software? In: Montanari, U., Sannella, D., Bruni, R. (eds.) TGC 2006. LNCS, vol. 4661, pp. 69–85. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75336-0_5

24. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: 2007 Future of Software Engineering, FOSE 2007, pp. 259–268. IEEE (2007)
25. Malek, S., Mikic-Rakic, M., Medvidovic, N.: A decentralized redeployment algorithm for improving the availability of distributed systems. In: Dearle, A., Eisenbach, S. (eds.) CD 2005. LNCS, vol. 3798, pp. 99–114. Springer, Heidelberg (2005). https://doi.org/10.1007/11590712_8
26. Namiot, D., Sneps-Sneppe, M.: On micro-services architecture. Int. J. Open Inf. Technol. **2**(9), 24–27 (2014)
27. Omicini, A., Ricci, A., Viroli, M.: Artifacts in the A&A meta-model for multi-agent systems. Auton. Agents Multi-Agent Syst. **17**(3), 432–456 (2008)
28. Ribino, P., Cossentino, M., Lodato, C., Lopes, S., Sabatucci, L., Seidita, V.: Ontology and goal model in designing bdi multi-agent systems. WOA@ AI* IA, **1099**, 66–72 (2013)
29. Rouvoy, R., et al.: MUSIC: middleware support for self-adaptation in ubiquitous and service-oriented environments. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 164–182. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02161-9_9
30. Rushby, J.: A safety-case approach for certifying adaptive systems. In: AIAA Infotech@ Aerospace Conference and AIAA Unmanned... Unlimited Conference, page 1992 (2009)
31. Sabatucci, L., Cossentino, M.: From means-end analysis to proactive means-end reasoning. In: Proceedings of 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, 18–19 May 2015
32. Sabatucci, L., Cossentino, M., Lodato, C., Lopes, S., Seidita, V.: A possible approach for implementing self-awareness in JASON. EUMAS **13**, 68–81 (2013)
33. Sabatucci, L., Lodato, C., Lopes, S., Cossentino, M.: Towards self-adaptation and evolution in business process. In: AIBP@ AI* IA, pp. 1–10. Citeseer (2013)
34. Sabatucci, L., Lodato, C., Lopes, S., Cossentino, M.: Highly customizable service composition and orchestration. In: Dustdar, S., Leymann, F., Villari, M. (eds.) ESOCC 2015. LNCS, vol. 9306, pp. 156–170. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24072-5_11
35. Sabatucci, L., Lopes, S., Cossentino, M.: A goal-oriented approach for self-configuring mashup of cloud applications. In: 2015 International Conference on Cloud and Autonomic Computing (ICCAC) (2016)
36. Sabatucci, L., Lopes, S., Cossentino, M.: Self-configuring cloud application mashup with goals and capabilities. Clust. Comput. **20**(3), 2047–2063 (2017)
37. Sabatucci, L., Ribino, P., Lodato, C., Lopes, S., Cossentino, M.: GoalSPEC: a goal specification language supporting adaptivity and evolution. In: Cossentino, M., El Fallah Seghrouchni, A., Winikoff, M. (eds.) EMAS 2013. LNCS (LNAI), vol. 8245, pp. 235–254. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-45343-4_13
38. Schmidt, D.C.: Model-driven engineering. IEEE Comput. Soc. **39**(2), 25 (2006)
39. Sommerville, I., et al.:: Software engineering. Addison-Wesley, Reading (2007)
40. Weyns, D., Georgeff, M.: Self-adaptation using multiagent systems. IEEE Softw. **27**(1), 86–91 (2010)
41. Weyns, D., Haesevoets, R., Helleboogh, A., Holvoet, T., Joosen, W.: The macodo middleware for context-driven dynamic agent organizations. ACM Trans. Auton. Adapt. Syst. (TAAS) **5**(1), 3 (2010)

42. Weyns, D., et al.: On patterns for decentralized control in self-adaptive systems. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems II. LNCS, vol. 7475, pp. 76–107. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35813-5_4
43. Zambonelli, F., Castelli, G., Mamei, M., Rosi, A.: Programming self-organizing pervasive applications with SAPERE. In: Zavoral, F., Jung, J., Badica, C. (eds.) Intelligent Distributed Computing VII. Studies in Computational Intelligence, vol. 511. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-01571-2_12