Lesson Learnt from Designing Self-Adaptive Systems with MUSA

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

[1] C.N.R., Istituto di Calcolo e Reti ad Alte Prestazioni, Palermo, Italy
[2] Dip. dell'Innovazione Industriale e Digitale
Università degli Studi di Palermo, Italy
{luca.sabatucci, massimo.cossentino}@cnr.it, 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 because of the self-adaptive nature of the system, the solution is not fixed at design time but it is a runtime outcome.
A new design process paradigm is needed to design such systems. In this paper, we present a retrospective analysis based on the results of three projects developed in the last five years with the middleware MUSA in order to identify specific features of the design process for supporting continuos change and self-adaptation.
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 organizations demand for a paradigm shift: from monolithic applications (that puts all user interfaces, business logics and data in a single process) toward applications that enables 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.

In this paper, we propose a retrospective analysis based on the use of MUSA (Middleware for User-Driven Service Adaptation) for developing self-adaptive systems. This study has been conducted on three different projects developed in the last years. The aim of this analysis is to explore the way in which design time and run-time draw near and intersect and which are the elements of the process involved in that. Finally, the results have been used for generalizing and understanding if, and under which conditions, the approach used with MUSA may be extended to other approaches to self-adaptation.

The rest of the paper is organized as follows: Section 2 illustrates some existing middleware for self-adaptive systems and introduce the need for systematic design approaches; Section 3 discusses the retrospective analysis on MUSA; in Section 4 the obtained results are discussed and in Section 5 we discuss them in order to generalize the approach; finally in Section 6 some conclusions are drawn.

2 Continuous Changes and Self-Adaptation

Complex distributed software requires continuous changes: it is a mix of continuous delivery and continuous integration. The three keywords are ‘frequent’, ‘reliable’ and ‘seamless’.

Automation is certainly one way to enable continuous changes. In particular we are interested in exploring the automation that supports continuous changes.

One of the characteristics of self-adaptive systems is to be designed for simplifying changing the behavior at run-time. In practice this category of autonomous systems is able of changing at run-time in order to either maintain or enhance their functions. This feature may be exploited for allowing continuous delivery and continuous integration without going offline. Now we want to focus on self-adaptation as the means to assure continuous changes to the running software. It is a matter of facts that many of the proposed approaches achieve general adaptation techniques by exploiting models-at-run-times and reflection.

This section is organized as follows: the first subsection focuses on different middleware, highlighting how they support different kinds of reflection. Subsequently, we look at the literature for understating which methodological support to be provided to developers. Finally, the need for a specific process raises a research question that motivates this work.

2.1 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, 20], agent-based [21, 10], or bio-inspired [26, 14].

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.

SAPERE (Self-aware Pervasive Service Ecosystems) [26] is inspired to natural ecosystems in order to model dynamism and decentralization in apervasive networks. In SAPERE various agents coordinate through spatially-situated and environment-mediated interactions, to serve their own individual 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. In this way, self-adaptation is not performed at an individual level, but it is rather an emerging feature of the system.

Kramer and Magee [7, 17] 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 goal evolution. The architecture is suitable for implementing self-adaptive system of type 2, able to deal with anticipated changes by selecting among pre-computed adaptation strategies.

MUSA (Middleware for User-driven Service Adaptation) [22] is a middleware for orchestrating distributed services according to unanticipated and dynamic user needs. Its main abstractions are the *Goal* and the *Capability*. They represent, respectively, what the system is expected to do and what the system knows to be able to do. Once goals and capabilities are specified, a Proactive Means-End Reasoning [21] associates system capabilities to the user goals for deducing possible *Solution Models* that can guarantee the desired final state (i.e. goal addressing). Therefore, MUSA is able to organize and orchestrate its components in order to actually reach the desired result.

The benefits of these middlewares are that they provide basic functionalities for rapid prototype 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 in order 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 own behavior for the future.

From a software engineering point of view using these middleware implies a methodological approach that moves design artifacts from the design-time to the run-time.

2.2 Need of a Process for Continuous Change

A promising approach to manage complexity in runtime environments and to implementing MAPE activities is to develop adaptation mechanisms that involves software models. This is referred to as *models@run.time* [6]. The idea is to extend the model produced using MDE approaches to runtime environment. Blair et al. emphasize the importance that software models (artifacts) may play at runtime, they also use the mechanism of reflection inducing that the necessary adaptation is performed at the model level rather than at the system level. In so doing, researchers in this field stop at artifact levels; they wish artifact produced were tied to the process used for creating them. However, at the best of our knowledge this is still a vision, an idea, and nothing has been really realized for joining the design phases and the runtime.

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 really implementing adaptation [1, 8].

As already said, requirements engineering has to deal with requirements that change at runtime as the result, for instance, of changing in the environment. Uncertainty and incompleteness are at the base of requirements engineering for self-adaptive systems [16, 9]. Some researchers investigated the use of a goal model for specifying behavior and requirements [18, 15] and for supporting the modeling of adaptation mechanism instead of implementing adaptation at runtime.

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 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 to make run-time changing and that the self-adaptive system itself supports the new development phase aiding, or better substituting, designers.

2.3 Research Question

This work focuses on the problem of developing a self-adaptive system by exploiting specific middleware from literature. The research question we raise in this paper is:

RQ: which are the characteristics of a design process for supporting self-adaptive middleware?

In order to investigate this topic we exploit the experience gained with MUSA, a self-adaptive middleware, and then we try to generalize some of the obtained results.

3 A Retrospective Analysis of MUSA

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

We selected MUSA [22] because we gained a practical experience of use, due to its adoption in several applications.

A MUSA application is not directly interfaced to the environment; instead, ad hoc sensors and actuators are utilized for this purpose. The capability is the abstraction that allows to specify not only the operations necessary for enacting the agent's decisions (actions on the environment, interactions) , but also when and how to use them for addressing a purpose. A capability represents an atomic and self-contained action the system may intentionally use to address a (partial) result.

In MUSA goals are run-time entities that represent the specification of its requirements. They are run-time entities in order to allow the system to be able to reason upon them. The framework exploits the GoalSPEC language [24] in order to facilitate the description of requirements. It is based on structured English and adopts a core grammar to which domain-specific terms can be added. It is automatically converted into a set of agent's belief statements (first order logic predicates). A solution model represents a contextual composition of capabilities that may be used for addressing a set of goals.

It is worth noting that in MUSA there is not a direct connection between goals and capabilities (indeed it is up to the system the responsibility to discover, at run-time, possible solution models). The only limitation is that both goal and capability specifications must be compliant with the same conceptualization of the world. In other words, it is necessary to establish an ontology over which to derive the semantic description of these elements. Despite the ontology is not explicitly a run-time entity, its concepts and predicates are, because they constitute the language that support the agents BDI model.

Empirical Study Design. In the last years MUSA has been adopted in various 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 analysis mainly considered the design activities, related to ontology, capabilities and goals. It focuses on artifacts that have been created and/or modified. Iterations have been deduced by considering artifact versioning.

As well as in the traditional requirement analysis, every MUSA project started with a good understanding of the domain by means of an ontology. The first variable to study in this analysis is the ontology. In MUSA, the ontology is necessary because it represents the common language for defining all the

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

Achronym	Type	App. Name	Description
IDS	Research Project	Innovative Documen 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.

entities-at-runtime that will be developed in the subsequent phases. The ontology definition exploits a POD diagram [12]. In particular the analysis focuses on two main aspects: the POD artifact and the effort spent on it.

A second point that is worth to be discussed concerns the definition of the services the system may employ in the emerging solution/architecture. In MUSA capabilities are run-time artifacts that are central for enacting solutions to address user's goals. This is the second variable we decided to include in our study. In particular the analysis focuses on two main aspects: the number of capability artifacts and the effort spent on decomposing the problem is smaller (atomic) services.

Finally, during the goal modeling phase, customer's requirements are translated into significant states of the system to be addressed. In some circumstances, in our projects, this activity implied a small revision of the ontology, and, consequentially, of the capabilities. This is the third variable of the study and includes the complexity of the goal model and the effort spent in revising goal specifications.

4 Interpretation of Results

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

The use of MUSA for developing and executing self adaptive systems implies to perform once a classical design phase. After that, the system is got on execution and every required change has to be handled while the system is running. In our approach that means several new iterations of design and execution activities. Some design activities are performed with the aid of agents that act "in the loop" of the MAPE actions. Which initial activities are modified or affected by the intervention of agents?

In order to answer these questions, it is necessary to provide some more details about how MUSA works. MUSA is based on collaborating agents and

Table 2. Empirical data obtained by retrospective analysis of three research projects in which MUSA has been adopted for developing a self-adaptive system.

Project	IDS	OCCP	Smart Travel
First Prototype			
type of change	first iteration	first iteration	first iteration
ontology size (only leaf concepts)	6	10	12
capability size	4	5	3
goal size	4	8	5
effort for ontology	10.00	30.00	7.00
effort for capability	30.00	70.00	40.00
effort for goals	14.00	7.00	100.00
total duration (to injection)	54.00	107.00	147.00
average solution size	4	5	15
number of solutions	1	1	5
Second Prototype			
type of change	bugfix+evolution	evolution	bugfix+evolution
ontology size	9	10	12
capability size	6	8	5
goal size	6	8	7
effort for ontology	7.00	10.00	7.00
effort for capability	21.00	40.00	25.00
effort for goals	7.00	7.00	14.00
total duration (to injection)	35.00	57.00	46.00
average solution size	6	6	15
number of solutions	3	9	5
Third Prototype			
type of change	evolution	bugfix+evolution	bugfix+evolution
ontology size	10	10	14
capability size	7	12	8
goal size	7	9	7
effort for ontology	3.00	7.00	14.00
effort for capability	17.00	50.00	20.00
effort for goals	1.00	4.00	1.00
time to injection	21.00	61.00	35.00
average solution size	6	7	18
number of solutions	6	18	5

artifacts [19]. Figure 2 depicts the main stakeholders, agents and artifacts involved in this process. The agent is the central element of the infrastructure. According to the classic vision, an agent is the owner of some capabilities, i.e. it can perceive the environment and act in order to change it. In addition MUSA agents know which capabilities they have and how to use them (thanks to the access to the *Capability Repository* artifact). The unique and shared design goal, common to all the agents, is to address users' run-time goals (i.e. requirements), when these are injected into the system.

When the user specifies a new set of goals to be addressed (by updating the *Specification* artifact), then the *solution explorer* agents are motivated to find one or more solution models. Given the perceived state of the world, the injected set of goals and the set of available capabilities, the problem (also referred as the Proactive Means-End Reasoning) is to find a complete composition of capabilities that may be used for addressing the goals.

Solution explorers collaborate, each exploiting its own knowledge, for exploring a space of solutions (the *Solution Graph* artifact), where states represent possible worlds they can reach by applying their capabilities in sequences. The algorithm is described in [21, 23].

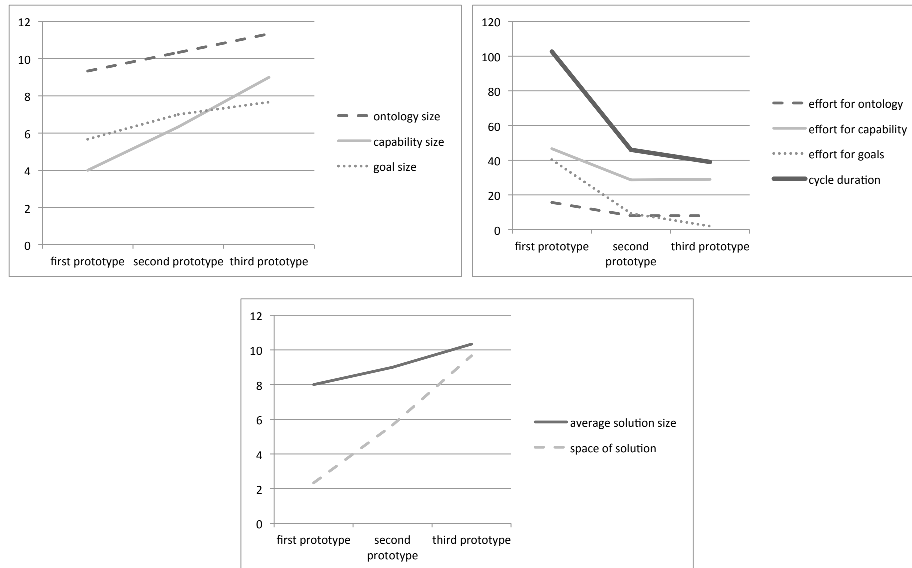

Fig. 1. Charts reporting average data, along three iterations, as extracted for the three projects. Top-left diagram shows the average increase of the complexity of the ontology, the capability model and the goal model. Top-right diagram shows the corresponding effort (in man hours) required to complete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

The definition of the domain is performed once and offline, whereas all new goals and capabilities, their definition and implementation, hence the core of the design phase, are produced/updated by the developer and the customer in conjunction with the service manager and the solution explorer while the system is already running.

This is not a mere automation of human activities, but instead a way for enhancing the system and releasing new versions while the system is running. Automation may be performed offline instead self-adaptation is performed online.

MUSA agents' social organizations are based on holons [13], an elegant and scalable means to guarantee autonomy, self-organization, distributed coordination, knowledge sharing and robustness. Whereas agents (with their capabilities) are the core bricks for producing basic functionalities, holons allow to dynamically compose more basic functionalities in a complex one by self-organizing several agents in multi-level architectures.

The *arch manager* agent is responsible for orchestrating the formation of a suitable hierarchy for the solution model (*Configuration Set* artifact). The hierarchical nature of holons allows to mix the top-down (goal to capability) recruitment approach, and the bottom-up service composition, that are both

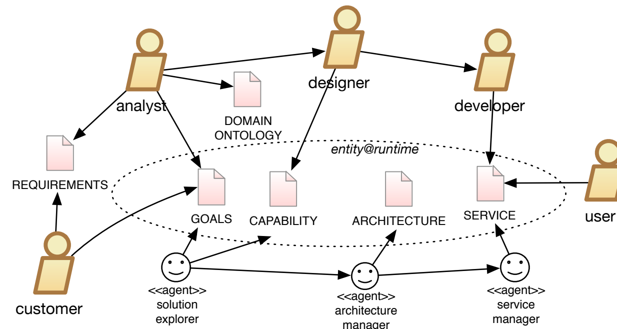

Fig. 2. The Human-Agent collaboration for the development of a MUSA application.

needed to realize complex functionalities. In either cases, some of the *service manager* agents enter the holon and become responsible for producing a part of the whole result. Coordination and knowledge sharing, within a holon, is respectively realized through the *Active Case* and the *Shared Context* artifacts.

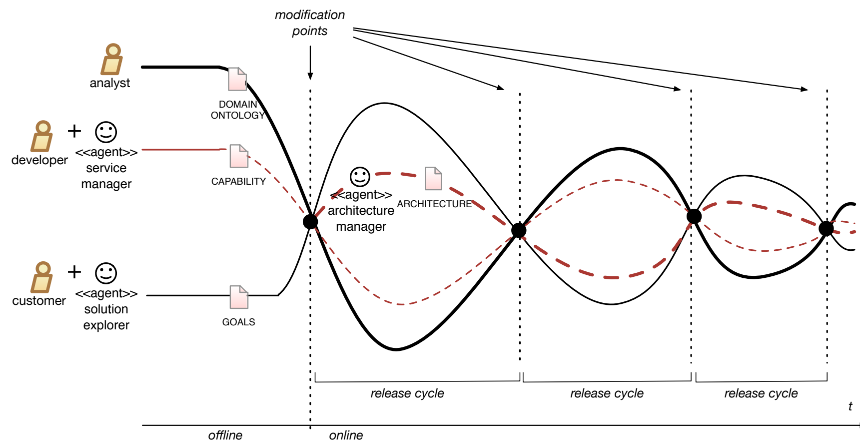

Fig. 3. Outcome of the retrospective analysis.

The retrospective analysis on the use of MUSA in three different projects (whose development has taken about five years) has given some interesting results that are summarized in Figure 3. The boundary between the offline and online design is highlighted by the first “modification point” - a modification

point is a moment in the system lifetime in which a new life cycle begins. The time interval between two different modification points is used by the system for embracing occurring changes in the operating condition and for releasing new configurations of the system. 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. Until this point the system is not still at work and the analyst produces a detailed representation of the domain of interest by means of a Problem Domain Ontology.

The developer designs a first set of capabilities the system has to own and the customer designs a first set of goals the system has to pursue using the right capabilities. These two activities are performed respectively, with the aid of the service manager agent and the solution explorer agent. The time spent in this first phase, obviously depends on the kind of problem to be faced, it is a classic design phase. After the first modification point, hence after the system is running and some changes are occurring, the system has to adapt, to be (re)designed online.

In other words, once the system is deployed and fired, agents collaborate in order to achieve the goals. They exploit the available capabilities and plan a solution (duty of the Solution Explorer agent). The solution is a set of capabilities that opportunely orchestrated may achieve the goals. Orchestration is left to the coordination occurring among agents. They are therefore involved in the construction of a dynamically built holonic architecture under the supervision of the Architecture Manager agent. While the system is running, the MUSA middleware permits the injection of new goals or capabilities. Sometimes even the domain ontology has to be evolved at runtime (in order to support the definition of new goals and capabilities). Once new goal/capabilities are injected, as already discussed before, the Solution Explorer agents look for the solution to the new goals or for the improvement of the solution for an existing goal. If a new solution is found (a new set of capabilities), the Architecture Manager agent proceeds with the construction of the new holons by inviting agents managing the selected capabilities in participating in the new piece of work.

From the analysis of the selected three projects we realized that after each modification point, each release takes less time and effort to be delivered and above all it implies less and less interactions among MUSA and the designers. Hence, the interleaving between design time an runtime increases from one release to another. Which is the explanation of this fact?

We may accept the following hypothesis: the tight interleaving is reached because the system modifies its behavior by evolving itself. Evolution consists in the fact that, every time a modification in the running/operating conditions (a new goal, a new requirement, a change in the environment or a change in the way the user uses the system) occurs, the system has to be endowed with new capabilities or goals and if necessary the ontology domain has to be refined. This second activity pertains the analyst and it is reasonable to think that, for each release time slot, it requires less and less time. In fact, for a specific domain context and a specific class of problem the knowledge about the domain is quite

complete after the first design phase, we think it is reasonable that only a few changes and refinements have to be introduced. So, this activity requires, for each release, less time and interactions between system and analyst.

As regards capabilities and goals, each time new capabilities are added and new goals are pursued, the system has acquired the ability to do more and diverse things, to recover and orchestrate several more services to be used. We suppose that, and this is confirmed by the analysis of projects data, after a certain number of releases, the knowledge on the environment becomes reasonably "stable" and each change is supported with less effort due to the fact that the system possesses all the capabilities for pursuing each kind of new goals. The system self-adapts. Fewer capabilities need to be added each time and new goals may be reached with less effort. After four or five modification points, design time and runtime activities converge in a system that is able to search for (and implement) the right functionality for facing each kind of change.

It is important to note that the evolution and consequent levels of self-adaptation is not the simple result of the use of MUSA but it mostly is the result of a way of conceiving the interaction among the involved parts of the systems. Analysts, developers, customers and the system itself operate in a synergistic fashion. Thus, the eventual injection of new goals and/or definition of new capabilities is carried out in a way that assure the consistency of changes with the initial designed systems. Handling the interactions using the middleware MUSA allows to vary the design process at runtime for managing continuous changes.

Summarizing, the study provides empirical evidences that:

1. A design process for MUSA includes a tight collaboration between human actors and agents of the system. In the specific case of MUSA human roles are: customer, analyst, designer and developer. Agents roles are: solution explorer, architect and service manager.
2. A set of run-time entities (goals and capabilities) represent the continuous data flow between engineers and agents (solution explorer and service manager) of the system.
3. Each run-time model is related to a separate life-cycle.
4. Frequent release cycles are performed in order to make the solution converge, and/or in order to broad the space of adaptation.

Concerning the RQ we set in Section 2, points 1-4 represent the basic characteristics that are necessary in a design process for supporting the development with MUSA. The next section tries to discuss whether the results of the retrospective analysis may apply to other middlewares and self-adaptation approaches than MUSA.

5 Discussion

A generalization is a broad statement that applies to many examples. In order to generalize what has been said for MUSA, we have to take in consideration also other middleware and approaches for self-adaptation.

Generalization. With the aim of providing a rationale for generalizing claims in Section 4, we have found pro e cons arguments. Both of them will be presented in the following, thus providing the reader with a complete landscape.

No, these results can not be generalized. MUSA works with a set of entities at run-time: goals, capabilities and the architecture of the solution. Changing the nature of run-time entities causes that the design process could change completely. For instance, in SeSaMe, the run-time entity is the group and the network topology: goals are embedded within nodes' roles. Another example is provided by SAPERE, in which each agent exists for a specific user's purpose, however the run-time entity is the eco-law, i.e. the rule specification that holds interaction among agents of the system. Taking these two middlewares as an example, the number and types of actors that collaborate in producing the system, as shown in Figure 2, could be no more valid.

Moreover, in MUSA a run-time model is related to a separate life-cycle because there is a neat separation of aspects among them. The use of a common ontology breaks the direct dependency among goals, capabilities and architecture. This could be different, in other middleware, in which a run-time model may depend on another one (run-time dependencies). In this case, designing or changing a model implies to design or revise the others. Life-cycles of Figure 3 could have one or more intermediate "integration points" where dependencies are solved.

We also have arguments supporting the generalization.

Yes, it is possible to generalize. A run-time model is inspired to the mechanism of reflection: i) inspect the code, ii) generate new code, or iii) change the existing code [25]. A run-time model represents a (semi) formal model expressed at a high-level of abstraction. Engineers design these models off-line and leverage them at run-time to drive system evolution [2]. The system uses them to organize its behavior. Hence, a run-time model documents itself. This consideration is quite general and can be easily extended to a large set of self-adaptive system that uses reflection [26, 5, 7, 11].

In practice, besides the kind of human actors and system agents, Figure 2 highlights the double nature of the run-time models: documentation artifacts and executable objects. This aspect of Figure 2 is quite general. Moreover, the central key of Figure 2 is the collaboration between human and agent (intended as an autonomous component of the system able to take decisions). The generalization of this architecture is that the enabler of this collaboration is the presence of run-time models. In other words, in self-adaptive systems, humans and the (components of the) system are both stakeholders of the resulting outcome.

Finally, analyzing the resulting software life-cycle, it is possible to claim that whatever middleware is used, after the first injection of run-time models, the system runs and produces a behavior. This means subsequent deploys are actually run-time modifications of the behavior. This because a run-time model generally exists until the system itself exists.

As already said, each run-time model represents a documentation artifact. A model artifact implies there is some kind of underlying design process that engineers used for producing and maintaining it. Even if it is not possible to take in consideration times and durations (because these are specific to MUSA), we can observe that each design thread in Figure 3 is a spiral life-cycle used for continuous prototyping the final result. In other words, Figure 3 represents a projection of a three dimensions diagram where many parallel spiral models develop along time, and meet each other in 'modification points' when run-time models change and the system evolve.

Concluding, the generalizable elements of a design process for self-adaptive middleware are:

1. the self-adaptive software becomes a stakeholder of itself;
2. run-time models represent documenting artifacts and executable objects;
3. the process is composed of threads, each following its own life-cycle;
4. each thread represents bidirectional collaboration between human stakeholders and the self-adaptive software;
5. the process has to consider some points in which engineers and/or the self-adaptive system apply evolutions.

Limits of this Analysis. The data extracted could be a bit bias because in these three projects, engineers developed both MUSA and its specialization for the specific domain. In the retrospective analysis the most complex part was separating the time required for fixing the middleware from the time required to implement 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, Figure 1 shows that the trend of the curves is quite regular. So we can hypothesize the sample is quite respectful of the reality.

6 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 [16, 9].

Self-adaptation deals with requirements that vary at run-time. Therefore it is important that requirements lend themselves to be dynamically observed, i.e., during execution. Middleware for self adaptation constitute good tools for easing

complex systems development and for providing a form of model@runtime but, a methodological supporting run-time continuos change still lacks.

In this paper, we illustrated the results of a retrospective analysis conducted on our middleware (MUSA) in order to identify the characteristics of a design process for developing self-adaptive systems. The analysis mainly highlighted that, using MUSA, supporting self-adaptive solutions implies a design process where humans and agents collaborate, goal and capabilities are the run-time entities that constitute the continuous data exchange between human and agents. Finally, the discussion about whether our approach may be generalized to others leads to the consideration that we cannot generalize if we change the kind of run-time entities and if the produced run-time models depend each others. Anyway, a run-time model is generally produced off-line and is used by the system for organizing its behavior, it documents itself thus fitting all the approaches that use reflection.

Moreover, the collaboration between humans and agents and the fact that a run-time model exists until the system is running guarantee system behavior modifications during subsequent releases.

References

1. J. Andersson, L. Baresi, N. Bencomo, R. de Lemos, A. Gorla, P. Inverardi, and T. Vogel. Software engineering processes for self-adaptive systems. In *Software Engineering for Self-Adaptive Systems II*, pages 51–75. Springer, 2013.
2. U. Aßmann, S. Götz, J.-M. Jézéquel, B. Morin, and M. Trapp. A reference architecture and roadmap for models@ run. time systems. In *Models@ run. time*, pages 1–18. Springer, 2014.
3. L. Baresi and C. Ghezzi. The disappearing boundary between development-time and run-time. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 17–22. ACM, 2010.
4. L. Baresi and S. Guinea. A3: self-adaptation capabilities through groups and coordination. In *Proceedings of the 4th India Software Engineering Conference*, pages 11–20. ACM, 2011.
5. L. Baresi, S. Guinea, and A. Shahzada. Sesame: towards a semantic self adaptive middleware for smart spaces. In *International Workshop on Engineering Multi-Agent Systems*, pages 1–18. Springer, 2013.
6. G. Blair, N. Bencomo, and R. B. France. Models@ run.time. *Computer*, 42(10):22–27, Oct 2009.
7. V. Braberman, N. D'Ippolito, J. Kramer, D. Sykes, and S. Uchitel. Morph: A reference architecture for configuration and behaviour self-adaptation. In *Proceedings of the 1st International Workshop on Control Theory for Software Engineering*, pages 9–16. ACM, 2015.
8. J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel. Towards a taxonomy of software change. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(5):309–332, 2005.
9. B. H. Cheng, R. De Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, et al. Software engineering for self-adaptive systems: A research roadmap. In *Software engineering for self-adaptive systems*, pages 1–26. Springer, 2009.

10. S.-W. Cheng. *Rainbow: cost-effective software architecture-based self-adaptation*. ProQuest, 2008.
11. S.-W. Cheng, D. Garlan, and B. Schmerl. Evaluating the effectiveness of the rainbow self-adaptive system. In *2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 132–141. IEEE, 2009.
12. M. Cossentino. From requirements to code with the passi methodology. *Agent-oriented methodologies*, 3690:79–106, 2005.
13. M. Cossentino, N. Gaud, V. Hilaire, S. Galland, and A. Koukam. Aspecs: an agent-oriented software process for engineering complex systems. *Autonomous Agents and Multi-Agent Systems*, 20(2):260–304, 2010.
14. J. L. Fernandez-Marquez, G. D. M. Serugendo, and S. Montagna. Bio-core: Bio-inspired self-organising mechanisms core. In *Bio-Inspired Models of Networks, Information, and Computing Systems*, pages 59–72. Springer, 2011.
15. H. J. Goldsby, P. Sawyer, N. Bencomo, B. H. Cheng, and D. Hughes. Goal-based modeling of dynamically adaptive system requirements. In *Engineering of Computer Based Systems, 2008. ECBS 2008. 15th Annual IEEE International Conference and Workshop on the*, pages 36–45. IEEE, 2008.
16. P. Inverardi. Software of the future is the future of software? In *International Symposium on Trustworthy Global Computing*, pages 69–85. Springer, 2006.
17. J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *Future of Software Engineering, 2007. FOSE'07*, pages 259–268. IEEE, 2007.
18. S. Liaskos, A. Lapouchnian, Y. Wang, Y. Yu, and S. Easterbrook. Configuring common personal software: a requirements-driven approach. In *13th IEEE International Conference on Requirements Engineering (RE'05)*, pages 9–18. IEEE, 2005.
19. A. Omicini, A. Ricci, and M. Viroli. Artifacts in the a&a meta-model for multi-agent systems. *Autonomous agents and multi-agent systems*, 17(3):432–456, 2008.
20. R. Rouvoy, P. Barone, Y. Ding, F. Eliassen, S. Hallsteinsen, J. Lorenzo, A. Mamelli, and U. Scholz. Music: Middleware support for self-adaptation in ubiquitous and service-oriented environments. In *Software engineering for self-adaptive systems*, pages 164–182. Springer, 2009.
21. L. Sabatucci and M. Cossentino. From Means-End Analysis to Proactive Means-End Reasoning. In *Proceedings of 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, May 18-19 2015.
22. L. Sabatucci, C. Lodato, S. Lopes, and M. Cossentino. Highly customizable service composition and orchestration. In *European Conference on Service-Oriented and Cloud Computing*, pages 156–170. Springer, 2015.
23. L. Sabatucci, S. Lopes, and M. Cossentino. A goal-oriented approach for self-configuring mashup of cloud applications. In *Cloud and Autonomic Computing (ICCAC), 2015 International Conference on*, 2016.
24. L. Sabatucci, P. Ribino, C. Lodato, S. Lopes, and M. Cossentino. Goalspec: A goal specification language supporting adaptivity and evolution. In *International Workshop on Engineering Multi-Agent Systems*, pages 235–254. Springer, 2013.
25. P. Sawyer, N. Bencomo, J. Whittle, E. Letier, and A. Finkelstein. Requirements-aware systems: A research agenda for re for self-adaptive systems. In *2010 18th IEEE International Requirements Engineering Conference*, pages 95–103. IEEE, 2010.
26. F. Zambonelli, G. Castelli, M. Mamei, and A. Rosi. Programming self-organizing pervasive applications with sapere. In *Intelligent Distributed Computing VII*, pages 93–102. Springer, 2014.