The Four Types of Self-adaptive Systems: A Metamodel

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

[1] ICAR-CNR, Palermo, Italy
{luca.sabatucci,massimo.cossentino}@icar.cnr.it
[2] DIID, Università degli Studi di Palermo, Palermo, Italy
valeria.seidita@unipa.it

Abstract. The basic ideas of self-adaptive systems are not a novelty in computer science. There are plenty of systems that are able of monitoring their operative context to take run-time decisions. However, more recently a new research discipline is trying to provide a common framework for collecting theory, methods, middlewares, algorithms and tools for engineering such software systems. The aim is to collect and classify existing approaches, coming from many different research areas. The objective of this work is providing a unified metamodel for describing the various categories of adaptation.

1 Introduction

Today's society always more depends on complex distributed software systems available 24 h and with minimal human supervision and maintenance effort during the operating phase. The more software systems grow in complexity and size, the more management automation, robustness, and reliability become central: it becomes essential to design and implement them in a more versatile, flexible, resilient and robust way.

In [17] authors discuss how an ambient intelligent system (AmI) plunges into the real world. The authors underline that, in such complex systems, the boundary between software and society blends and often disappears. The social environment is enriched with artificial intelligence to support humans in their everyday life. The IBM manifesto of autonomic computing [11] suggests a promising direction for facing software complexity through self-adaptation.

A self-adaptive system is a system with the ability to autonomously modify its behavior at run-time in response to changes in the environment [5,7,21]. The vision of a computing system that can manage itself is fascinating [5,7]: to modify the behavior at run-time for maintaining or enhancing its functions [5]. This vision has deep roots in several research fields, as for instance, artificial intelligence, biologically inspired computing, robotics, requirements/knowledge engineering, control theory, fault-tolerant computing, and so on. In the last decade, the vast and heterogeneous number of works concerning self-adaptation investigated several aspects of the problem, for instance, specific architectures for implementing adaptive control loops [14], self-organizing paradigms [1], adaptive requirements [6] and so on. However, to date, many of these problems remain significant intellectual challenges [5,7]. For instance, general purpose software engineering approaches are still missing for the provision of self-adaptation [5]. The long-term objective is to establish the foundations for the systematic development of future generations of self-adaptive systems.

This work resembles existing approaches for systematically engineering self-adaptive system and proposes a unified metamodel of the four types of adaptation. The objective is to provide a framework for identifying and classifying smart systems according to their self-adaptive properties. A metamodel supports this framework for aiding the designer to choose the most appropriate category of systems, depending on the problem statements. The description reports the main components and some illustrative case studies for each type of adaptation.

The paper is structured as follows: Sect. 2 analyzes definitions of self-adaptive systems in state of the art. Section 3 presents the unified metamodel and describes the four categories in details. Finally, Sect. 4 reports the conclusions.

2 Related Work

In last decade, the definition of adaptation has been deliberately generic to gather many sub-fields under a common umbrella and produce interesting synergy. However, this trend has generated some sub-definitions with sometimes significant differences. For instance, in [5,7] a self-adaptive system can modify its behavior in response to changes in the environment. For the models@run.time community [12], a dynamically adaptive system (DAS) can be conceptualized as a dynamic software product line in which variabilities are bound at runtime for improving the quality of service (QoS).

Unifying different definitions is required. Salehie and Tahvildari [20] identify two categories of self-adaptation based on impact (the scope of system effects) and cost factors (in terms of time, resources and complexity). The weak adaptation mainly involves modifying parameters using pre-defined static mechanisms (limited impact/low cost). Conversely, the strong adaptation deals with high-cost/extensive impact actions such as adding, replacing, removing components.

In [16], the authors provide a classification scheme for four categories of self-adaptive systems: *Type 1* consists in anticipating both changes and the possible reactions at design-time: the system follows a behavioral model that contains decision-points. For each decision point, the solution is immediately obvious given the current perceptions and the acquired knowledge about the environment.

Type 2 consists of systems that own many alternative strategies for reacting to changes. Each strategy can satisfy the goal, but it has a different impact on some non-functional requirement. Selecting the best strategy is a run-time operation based on the awareness of the different impact towards these external aspects. Typically the decision is taken by balancing trade-offs between alternatives, based on the acquired knowledge about the environment.

Type 3 consists of systems aware of its objectives and operating with uncertain knowledge about the environment. It does not own pre-defined strategies but it rather assemblies ad-hoc functionalities according to the execution context.

Type 4 is inspired by biological systems that are able of self-modifying their specification when no other possible additions or simple refinements are possible.

In [2], authors face the same point, but from a requirement engineering perspective. The premise is that requirement engineering task is to determine the kinds of input of a system and the possible responses to these inputs. Therefore they identify four types of requirement engineering activities concerning a self-adaptive system.

Level 1 activities are done by humans and resemble the traditional RE activities. The analysts determine all the possible domains to be considered by the system (inputs) and all the possibility system functionalities (reactions).

The system executes *Level 2*'s activities. Whereas the analysts have determined a set of possible behaviors (i.e. reactions), the system can identify the functionality to execute next, when the environment does not match any of the input domains.

Conversely, *Level 3* activities are done by humans for implementing the decision-making procedure that allows the system to apply level 2. Level 3 often includes a meta-level reasoning, that exploits determined program-testable correspondences to environmental changes that trigger adaptation.

Implementing the adaptation mechanism (i.e. the feedback loop) is a *Level 4* activity, for which humans are responsible.

3 The Proposed Metamodel

A smart system is often immersed in a pre-existing world (or environment) populated by objects and persons it interacts with, it influences and is influenced. Boundaries between the software realizing the smart system and the environment are becoming lighter, more and more, and they are almost disappearing. All this significantly affects the design of smart systems. In this section, we present a framework for identifying and illustrating which may be the minimal set of elements (Fig. 1) a system as to own for being classified self-adaptive of the four cited types.

The metamodel in Fig. 1 is composed of two parts: the first part includes all the generic elements of a smart system, whereas the second part embraces all the elements implementing the different types of self-adaptation. The first part comes from a previous study of some authors of this paper [17], where a set of abstractions for representing smart systems in an Ambient Intelligent context has been explored and experimented. All the elements in the second part have been identified reviewing the literature and definitions about managing and designing self-adaptive systems.

We argue that, in general, abstractions representing a smart system are mainly the environment and all the entities it is composed of. There may be, passive, dumb and smart entities, all of them present a state that may be perceived and changed during the running phase of the system. Smart entities are cognitive, intentional and rational entities, able to perform actions according to the principle of rationality. Dumb and passive entities are elements of the environment, the former are resources (software applications, physical devices, ...) and the latter are simply objects in the environment such as physical or digital objects; both of them are part of the environment and influence the actions of the smart system.

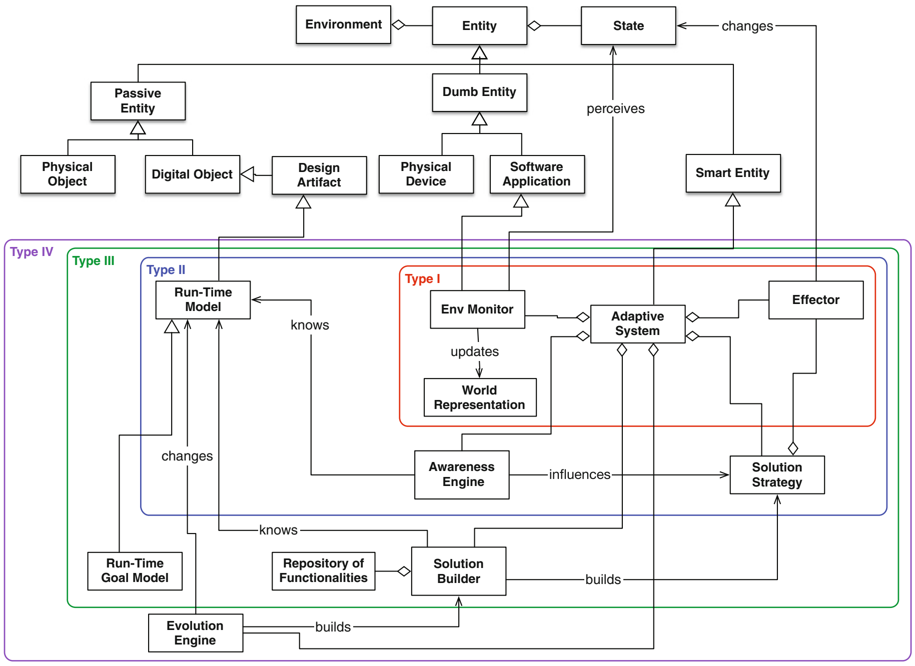

Fig. 1. Portion of metamodel that describes the four types of self-adaptive systems.

The second part of the metamodel shows all the elements of a self-adaptive system. As it may be deduced by the definitions, the set of elements for each type is contained in the higher ones. A smart system belongs to the first type of self-adaptation if it owns a kind of smart entity, the *Adaptive System* composed of the elements necessary to know the environment and to act on it. The second type principally requires an *Awareness Engine* qualified to know the model at run-time of the system and to change or influence the solution strategy built at design time. The third type involves a more complex element that is the *Solution Builder*, this is able to build a completely new solution strategy by using a repository of established functionalities. Finally, the fourth type requires the *Evolution Engine* that is able to set up a *Solution Builder*.

In the following, each type is broadly described. It is important to note that *Design Artifact, Run-Time Model, Awareness Engine, Solution Builder, Evolution Engine* are not classic elements of a problem domain you may find in a metamodel but are strictly related to the software solution. This is to stress the fact that self-adaptive systems cannot be conceived or designed if not as an alive part of the domain.

3.1 Type I

Adaptation of *Type I* is the simplest implementation of smart systems. It arises from a deep analysis of the domain for analyzing all the possible changes the system will observe and react to. The design activity includes the study of all the possible reactions the system will enact. The design leads to the definition of a behavioral model that contains decision-points: a decision point is analogous to a 'if...then...else' statement, whereas conditions typically depend on perceptions of the execution environment.

Problem Statement. The Adaptation of *Type I* is a good choice when:

- it is necessary a smart system able to operate in different ways, according to the state of the environment, acquired by perceptions;
- changes mainly affect observable attributes of the environment;
- it is possible to anticipate (at design-time) all the environment changes that are of interest for the system;
- dealing with uncertainty is not central in the implementation.

System Description. An Adaptive System of Type I is a specific class of artificial system, i.e. a smart entity that operates in an environment. An Adaptive System of Type I owns effectors for changing the state of the environment and perceptors (Env Monitor) for acquiring the current state of the environment. Decisions points are rules that connect knowledge and perception (cause) to an effector (consequence).

Known Usage. An illustrative scenario for Type I is the Robotic Navigation System presented in [5]. The scenario illustrates autonomous control software system of unmanned vehicles (UVs). The adaptation system must consider the regular traffic environment, including the infrastructures and other vehicles. A cause of adaptation can be an unexpected obstacle (people crossing the road). To this aim, it is necessary to monitor the environment (perceptors) and to detect possible obstacles in front of the vehicle. An example of a cause-consequence rule is: IF an obstacle is in front of the vehicle THEN maneuver around the obstacle for avoiding a collision.

Another example is a smart information system for airports [15]. A cause of adaptation occurs when a flight delays. According to the importance of the delay, the system may select different actions: inform the traveler, rebook the flight as soon as possible, book a hotel and re-plan the flight the next day.

Limits. Do not use Type I:

- if the system decision must deal with non-functional aspects; for instance, to rebook a service as soon as possible, or to change the itinerary for maintaining the traveler's satisfaction.
- if a flexible strategy for changing user's goals is required.

3.2 Type II

Adaptation of *Type II* consists of complex systems that offer many alternative strategies for addressing the same goal. Different strategies encompass different operative situations and provide a different impact on the non-functional requirements. This situation requires revising the traditional analysis of requirements phase for including the detailed exploration of a large problem space. Selecting the contextual strategy to apply is a run-time decision that considers complex trade-offs among the state of the environment and the desired quality of service. This decision requires a higher level of knowledge than Type I. It must include the awareness of functional/non-functional requirements, and the different way each operation impacts on them.

Problem Statement. The Adaptation of *Type II* is a good choice when:

- it is necessary a smart system that can detect and recover run-time deviations between behavior and requirements;
- it is not possible to anticipate all the environment changes at design-time, and therefore it is preferable to discover them at run-time;
- changes can also affect requirements;
- it is necessary to incorporate a degree of uncertainty in requirements.

System Description. An Adaptive System of Type II contains a set of effectors and perceptors as well as the Type I and extends it with a set of solution strategies and an awareness engine. A solution strategy is a plan which actions are the monitors and effectors. A solution strategy (or its components) is linked to some functional requirement that motivates its execution. Moreover, the solution strategy (or its components) may also be related to one or more non-functional requirements, describing the kind of expected impact of its execution. This complex model is often a run-time artifact that the Awareness Engine acquires and manages.

Known Usage. An example of Type II adaptation is contained in the scenario of London Ambulance Service (LAS) [10]. According to official recommendations, such a system must involve a full process of consultation between management, staff, trade union representatives and the Service's information technology advisers. Authors have identified many solutions for the main goal (to respond emergency calls within a specific time). A solution is composed of many tasks, each one addressing a specific subgoal. The system follows an adaptation of Type II because it may enact alternative functions. The system reasons on which task to select (and therefore which is the best solution) according to quality aspects, for instance, the time, the efficiency and the reliability. Authors used a goal model [9] for depicting the problem requirements together with tasks and quality aspects. This goal model is implemented as a run-time model the system receives as an input for taking contextual decisions.

Another example comes from dynamic workflow execution engines. In [4] the authors provide a workflow execution engine where tasks are related to abstract services. Each abstract service is the high-level description of many actual services provided by different providers. A real service owns some quality of services. The execution of the workflow requires selecting among alternative concrete services. The choice is to optimize one of the global quality of service assets [8]: different solutions occur if the user decides to optimize the total cost or the total time to complete.

Limits. Do not use Type II:

– if user's functional/non-functional requirements are dynamics;
– if you desire a flexible way for extending system functionalities.

3.3 Type III

Adaptation of *Type III* represents an advanced implementation of a smart system that is instructed with a set of basic functionalities. The system can use for assembling ad-hoc behaviors not contained in any of the predefined solution strategies. This kind of system is particularly suitable for working with uncertain knowledge about the environment and the requirements.

Problem Statement. The Adaptation of *Type III* is a good choice when:

– the problem domain is not entirely anticipated at design-time,
– requirements evolve frequently due to business rules or societal norms;
– it is necessary to assemble ad-hoc functionalities on the fly;
– system functionalities are open to third-party providers and dynamics (depending, for instance, from network conditions).

System Description. An Adaptive System of Type III owns the core characteristics of Type II, but it supports a new component, called Solution Builder. This element can access to a (dynamic) repository of functionalities and build one or more new solution strategies for addressing an unanticipated problem. Selecting among many solutions frequently implies an optimization phase that also considers non-functional aspects.

Known Usage. An example of Type III adaptation is a smart travel assistance system [18]. Such a system can compose a trip itinerary according to user's preferences and create travel service as the composition of multiple atomic services (flight booking, hotel reservation, local transportation tickets and so on). The system is also able to monitor either possible problems during the journey, or changes in user's preferences, and to optimize the travel itinerary consequently.

The new frontier of service composition over the cloud is called cloud mashup. The adaptive scenario –described in [19]– includes on-demand, on-the-fly application mashup for addressing a set of designer's goals. The customer does not provide goals, so the designer can not incorporate them into the orchestration model. An adaptation system of type III allows injecting goals at run-time. Consequently, the system builds a new mashup by aggregating existing cloud services, according to availability, cost and reliability [13].

Limits. Do not use Type III:

– for real-time systems;
– if the system should be able of inspecting itself and autonomously evolving its functionality.

3.4 Type IV

Adaptation of *Type IV* is the higher level of a smart system that is able of inspecting itself, learning from experience and self-modifying its specification. They are designed to afford the worst cases of adaptation: when the system does not own suitable actions/strategy to be used, and it is not capable of generating any one. In this case, the system is able of revising its run-time model, thus to produce a new version of the software. In this category of adaptation, it is more appropriate to refer to evolution. Indeed these systems are inspired by biological systems that own the ability to cope with environment variance by genetic changes.

Problem Statement. The Adaptation of Type IV is a good choice:

– when developers deal with incomplete information about the highly complex and dynamic environment, and, consequently, incomplete information about the respective behavior that the system should expose;
– the system must be able of interpreting incomplete run-time models, and applying –when necessary– changes to those, in order to regulate its behavior;
– the system must be able of generating, at the best of its possibilities, a suitable strategy even when some basic functionality lacks.

System Description. Whereas in classical conception of a self-adaptive system (Type I, II, and III) the system can modify its behavior according to the specifications and to the environment changes, the self-adaptive systems of Type IV are also able of changing their specification. It may be considered as a strong form of learning. This could include some run-time technique for validating the new specification, performing, when necessary, possible trade-off analysis between several potentially conflicting goals.

Known Usage. To the best of our knowledge, there are not popular examples of self-adaptive systems of Type IV.

A case study that could benefit from this kind of adaptation could be a smart firewall [5]. It is a system able to respond to cyber-attacks, but it can not possibly know all attacks in advance since malicious actors develop new attack types all the time. So far, this kind of systems is yet a challenge for artificial intelligence and computer science. It implies a high level of self-awareness and the ability to learn, to reason with uncertainty and to generate new code for coping with unexpected scenarios.

4 Conclusions

All the types of self-adaptive systems share a shift of some design decisions towards run-time in order to improve the control over the behavior.

In practice, the reader has to focus on run-time activities and in particular on the decision-making process. This latter is the algorithm/technique used for directing the system behavior.

This paper proposes a framework for classifying systems and their self-adaptive attributes by means of a set of abstractions grouped into a metamodel. The metamodel offers the designer the possibility to select the right elements related to the chosen self-adaptive type. From a designer point of view, the power of this metamodel is that it considers the smart system and the environment it operates in strictly tied each other: the environment is part of the software solution and in the same way the system at run-time, with its design artifacts, algorithms and so on, is part of the environment thus realizing the feedback loop regulating all the activities of a self-adaptive system [3]. The monitor senses the environment and collects relevant data and events for future reference. The analyzer compares data in order to evaluate differences between the actual and the expected behavior. The planner uses these data for taking decisions about the behavior to be executed. Finally, the execute (or act) module applies the planned decisions through its effectors.

Moreover, in order to classify a smart system according to the type of adaptation, the following guideline focuses on the kind of perception ability and decision-making process. If the run-time activity is the enactment of a set of hard-coded actions (selected and/or configured according to the operative context), then the adaptation is of Type I. If the system owns a set of pre-defined strategies (each strategy is an aggregation of actions) and if the strategy is selected and/or configured at run-time, according to quality aspects, then the adaptation is of Type II. If the system is able of assembling a new strategy at run-time, then the adaptation is of Type III. If the system can modify its run-time models for generating new functions, then the system is of Type IV.

References

1. 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)
2. Berry, D.M., Cheng, B.H. Zhang, J.: The four levels of requirements engineering for and in dynamic adaptive systems. In: 11th International Workshop on Requirements Engineering Foundation for Software Quality (REFSQ), p. 5. (2005)

3. Brun, Y., Marzo Serugendo, G., Gacek, C., Giese, H., Kienle, H., Litoiu, M., Müller, H., Pezzè, M., Shaw, M.: Engineering self-adaptive systems through feedback loops. In: Cheng, B.H.C., Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-adaptive Systems, pp. 48–70. Springer, Heidelberg (2009)
4. Casati, F., Ilnicki, S., Jin, L.-J., Krishnamoorthy, V., Shan, M.-C.: eFlow: a platform for developing and managing composite e-services. In: Proceedings of the Academia/Industry Working Conference on Research Challenges, pp. 341–348. IEEE (2000)
5. Cheng, B.H.C., Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., et al.: Software engineering for self-adaptive systems: a research roadmap. In: Cheng, B.H.C., Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-adaptive Systems. Springer, Heidelberg (2009)
6. Dalpiaz, F., Giorgini, P., Mylopoulos, J.: Adaptive socio-technical systems: a requirements-based approach. Requir. Eng. **18**(1), 1–24 (2013)
7. Lemos, R., Giese, H., Müller, H.A., Shaw, M., Andersson, J., Litoiu, M., Schmerl, B., Tamura, G., Villegas, N.M., Voge, T., et al.: Software engineering for self-adaptive systems: a second research roadmap. In: Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-adaptive Systems II, pp. 1–32. Springer, Heidelberg (2013)
8. C. Di Napoli, D. Di Nocera, and S. Rossi. Computing pareto optimal agreements in multi-issue negotiation for service composition. In: Proceedings of the International Conference on Autonomous Agents and Multiagent Systems, pp. 1779–1780. International Foundation for Autonomous Agents and Multiagent Systems (2015)
9. I. Jureta, A. Borgida, N. A. Ernst, and J. Mylopoulos. Techne: Towards a new generation of requirements modeling languages with goals, preferences, and inconsistency handling. In: RE, pp. 115–124 (2010)
10. Jureta, I.J., Borgida, A., Ernst, N.A., Mylopoulos, J.: The requirements problem for adaptive systems. ACM Trans. Manag. Inf. Syst. (TMIS) **5**(3), 17 (2015)
11. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer **36**(1), 41–50 (2003)
12. Morin, B., Barais, O., Jezequel, J.-M., Fleurey, F., Solberg, A.: Models@ run.time to support dynamic adaptation. Computer **42**(10), 44–51 (2009)
13. Napoli, C.D., Sabatucci, L., Cossentino, M., Rossi, S.: Generating and instantiating abstract workflows with QOS user requirements. In: Proceedings of the 9th International Conference on Agents and Artificial Intelligence (2017)
14. Patikirikorala, T., Colman, A., Han, J., Wang, L.: A systematic survey on the design of self-adaptive software systems using control engineering approaches. In: ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), pp. 33–42 (2012)
15. Qureshi, N.A., Jureta, I.J., Perini, A.: Requirements engineering for selfadaptive systems: core ontology and problem statement. In: International Conference on Advanced Information Systems Engineering, pp. 33–47. Springer (2011)
16. Qureshi, N.A., Perini, A., Ernst, N.A., Mylopoulos, J.: Towards a continuous requirements engineering framework for self-adaptive systems. In: First International Workshop on Requirements@ Run.Time (RE@ RunTime), pp. 9–16. IEEE (2010)
17. Ribino, P., Cossentino, M., Lodato, C., Lopes, S., Seidita, V.: Requirement analysis abstractions for AmI system design. J. Intell. Fuzzy Syst. **28**(1), 55–70 (2015)

18. Sabatucci, L., Cavaleri, A., Cossentino, M.: Adopting a middleware for self-adaptation in the development of a smart travel system. In: Pietro, G., Gallo, L., Howlett, R.J., Jain, L.C. (eds.) Intelligent Interactive Multimedia Systems and Services 2016, pp. 671–681. Springer, Cham (2016)
19. Sabatucci, L., Lopes, S., Cossentino, M.: A goal-oriented approach for self-configuring mashup of cloud applications. In: International Conference on Cloud and Autonomic Computing (ICCAC), pp. 84–94. IEEE (2016)
20. Salehie, M., Tahvildari, L.: Self-adaptive software: landscape and research challenges. ACM Trans. Auton. Adapt. Syst. (TAAS) **4**(2), 14 (2009)
21. Whittle, J., Sawyer, P., Bencomo, N., Cheng, B.H., Bruel, J.-M.L Relax: incorporating uncertainty into the specification of self-adaptive systems. In: 2009 17th IEEE International Requirements Engineering Conference, pp. 79–88. IEEE (2009)