Chapter 8
Runtime Evolution of Highly Dynamic Software

Hausi Müller and Norha Villegas

Summary. Highly dynamic software systems are applications whose operations are particularly affected by changing requirements and uncertainty in their execution environments. Ideally such systems must evolve while they execute. To achieve this, highly dynamic software systems must be instrumented with self-adaptation mechanisms to monitor selected requirements and environment conditions to assess the need for evolution, plan desired changes, as well as validate and verify the resulting system. This chapter introduces fundamental concepts, methods, and techniques gleaned from self-adaptive systems engineering, as well as discusses their application to runtime evolution and their relationship with off-line software evolution theories. To illustrate the presented concepts, the chapter revisits a case study conducted as part of our research work, where self-adaptation techniques allow the engineering of a dynamic context monitoring infrastructure that is able to evolve at runtime. In other words, the monitoring infrastructure supports changes in monitoring requirements without requiring maintenance tasks performed manually by developers. The goal of this chapter is to introduce practitioners, researchers and students to the foundational elements of self-adaptive software, and their application to the continuos evolution of software systems at runtime.

DOI 10.1007/978-3-642-45398-4_8, © Springer-Verlag Berlin Heidelberg 2014

8.1 Introduction

Software evolution has been defined as the application of software maintenance actions with the goal of generating a new operational version of the system that guarantees its functionalities and qualities, as demanded by changes in requirements and environments [170, 598]. In the case of continuously running systems that are not only exposed frequently to varying situations that may require their evolution, but also cannot afford frequent interruptions in their operation (i.e., 24/7 systems), software maintenance tasks must be performed ideally while the system executes, thus leading to *runtime software evolution* [598]. Furthermore, when changes in requirements and environments cannot be fully anticipated at design time, maintenance tasks vary depending on conditions that may be determined only while the system is running.

This chapter presents runtime evolution from the perspective of *highly dynamic software systems*, which have been defined as systems whose operation and evolution are especially affected by uncertainty [646]. That is, their requirements and execution environments may change rapidly and unpredictably. Highly dynamic software systems are context-dependent, feedback-based, software intensive, decentralized, and quality-driven. Therefore, they must be continually evolving to guarantee their reliable operation, even when changes in their requirements and context situations are frequent and in many cases unforeseeable. Müller et al. have analyzed the complexity of evolving highly dynamic software systems and argued for the application of evolution techniques at runtime [621]. They base their arguments on a set of problem attributes that characterize feedback-based systems [622]. These attributes include (i) the uncertain and non-deterministic nature of the environment that affects the system, and (ii) the changing nature of requirements and the need for regulating their satisfaction through continuous evolution, rather than traditional software engineering techniques.

Control theory, and in particular feedback control, provides powerful mechanisms for uncertainty management in engineering systems [625]. Furthermore, a way of exploiting control theory to deal with uncertainty in software systems is through self-adaptation techniques. Systems enabled with self-adaptive capabilities continuously sense their environment, analyze the need for changing the way they operate, as well as plan, execute and verify adaptation strategies fully or semi-automatically [177, 221]. On the one hand, the goal of software evolution activities is to extend the life span of software systems by modifying them as demanded by changing real-world situations [595]. On the other hand, control-based mechanisms, enabled through self-adaptation, provide the means to implement these modifications dynamically and reliably while the system executes.

Rather than presenting a comprehensive survey on runtime software evolution and self-adaptive systems, this chapter introduces the notion of runtime software evolution, and discusses how foundational elements gleaned from self-adaptation are applicable to the engineering of runtime evolution capabilities. For this we organized the contents of this chapter as follows. Section 8.2 describes a case study, based on dynamic context monitoring, that is used throughout the chapter to explain the presented concepts. Section 8.3 revisits traditional software evolution to introduce the need for applying runtime software evolution, and discusses selected aspects that may be taken into account when deciding how to evolve software systems dynamically. Section 8.4 characterizes dimensions of runtime software evolution, and discusses Lehman's laws in the context of runtime evolution. Section 8.5 introduces the application of feedback, feedforward, and adaptive control to runtime software evolution. Sections 8.6 and 8.7 focus on foundations and enablers of self-adaptive software that apply to the engineering of runtime software evolution, and Section 8.8 illustrates the application of these foundations and enablers in the case study introduced in Section 8.2. Section 8.9 discusses selected self-adaptation challenges that deserve special attention. Finally, Section 8.10 summarizes and concludes the chapter.

8.2 A Case Study: Dynamic Context Monitoring

As part of their collaborative research on self-adaptive and context-aware software applications, researchers at University of Victoria (Canada) and Icesi University (Colombia) developed SMARTERCONTEXT [822, 895–898, 900]. SMARTERCONTEXT is a service-oriented context monitoring infrastructure that exploits self-adaptation techniques to evolve at runtime with the goal of guaranteeing the relevance of monitoring strategies with respect to changes in monitoring requirements [894].

In the case study described in this section, the SMARTERCONTEXT solution evolves at runtime with the goal of monitoring the satisfaction of changing quality of service (QoS) contracts in an e-commerce scenario [822]. Changes in contracted conditions correspond to either the addition/deletion of quality attributes to the contracted service level agreement (SLA), or the modification of desired conditions and corresponding thresholds. Suppose that an online retailer and a cloud infrastructure provider negotiate a performance SLA that specifies *throughput*, defined as the time spent to process a purchase order request ($ms/request$), as its quality factor. Suppose also that a first version of SMARTERCONTEXT was developed to monitor the variable relevant to the throughput quality factor (i.e., processing time per request). Imagine now that later the parties renegotiate the performance SLA by adding *capacity*, defined in terms of bandwidth, as a new quality factor. Since SMARTERCONTEXT has been instrumented initially to monitor processing time only, it will have to evolve at runtime to monitor the system's bandwidth. Without these runtime evolution capabilities, the operation of the system will be compromised until the new monitoring components are manually developed and deployed.

SMARTERCONTEXT relies on behavioral and structural self-adaptation techniques to realize runtime evolution. Behavioural adaptation comprises mechanisms that tailor the functionality of the system by modifying its parameters or business logic, whereas structural adaptation uses techniques that modify the system's software architecture [899]. SMARTERCONTEXT implements behavioral adaptation by modifying existing monitoring conditions or adding new context types and reasoning rules at runtime, and structural adaptation by (un)deploying context gatherers and context processing components. All these operations are realized without requiring the manual development or deployment of software artifacts, and minimizing human intervention.

8.3 Assessing the Need for Runtime Evolution

The need for evolving software systems originates from the changing nature of system requirements and the changing environment that can influence their accomplishment by the system [123]. Indeed, as widely discussed in Chapter 1, changing requirements are inherent in software engineering. For example, in the case of SMARTERCONTEXT the need for generating new versions of the system arises from the renegotiation of contracted SLAs, which implies changes in monitoring requirements. Several models have been proposed to characterize the evolution process of software systems [590]. In particular, the *change mini-cycle* model proposed by Yau et al. defines a feedback-loop-based software evolution process comprising the following general activities: change request, analysis, planning, implementation, validation and verification, and re-documentation [936]. Figure 8.1 depicts the flow among the general activities of the change mini-cycle process model. These activities were proposed for off-line software evolution, which implies the interruption of the system's operation.

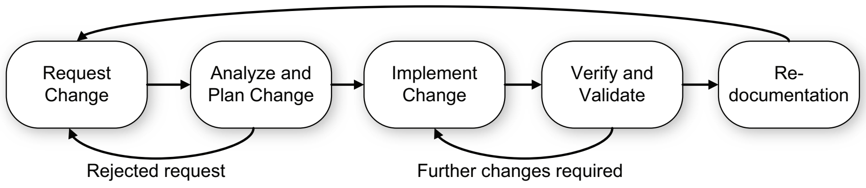


Fig. 8.1: Activities of the *change mini-cycle* process model for software evolution [936] (adapted from [590]). This model implements a feedback loop mechanism with activities that are performed off-line.

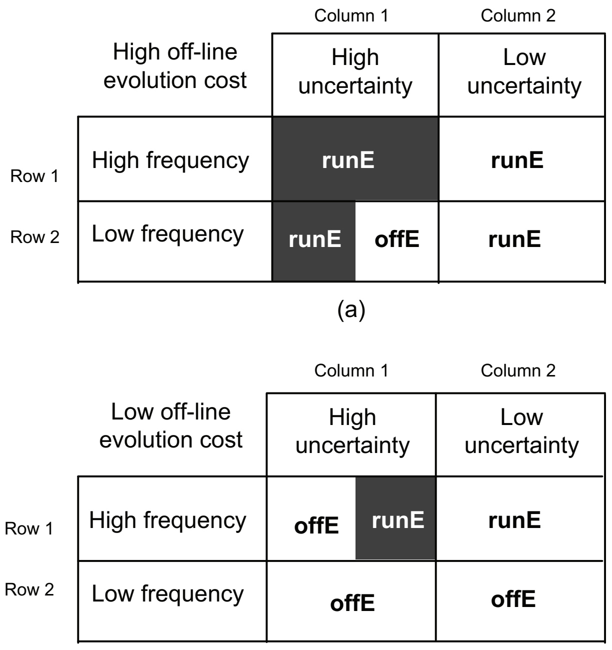
We define *off-line software evolution* as the process of modifying a software system through actions that require intensive human intervention and imply the interruption of the system operation. We define the term *runtime software evolution* as the process of modifying a software system through tasks that require minimum human intervention and are performed while the system executes. This section characterizes software evolution from a runtime perspective.

The need for evolving software systems emerges from changes in environments and requirements that, unless addressed, compromise the operation of the system. The need for evolving software systems *at runtime* arises from the frequency and uncertainty of these changes, as well as the cost of implementing off-line evolution. In the context of software evolution, we define *frequency* as the number of occurrences of a change event per unit of time that will require the evolution of the system. For example, the number of times an SLA monitored by SMARTERCONTEXT is modified within a year. We define *uncertainty* as the reliability with which it is possible to characterize the occurrence of changes in requirements and execution environments. The level of uncertainty in a software evolution process depends on the deterministic nature of these changes. That is, the feasibility of anticipating their frequency, date, time, and effects on the system. In the case of SMARTERCONTEXT, changes in SLAs will be less uncertain to the extent that it is possible to anticipate the date and time contract renegotiation will occur, as well as the aspects of the SLA that will be modified including quality factors, metrics and desired thresholds. The cost of implementing off-line evolution can be quantified in terms of metrics such as system's size, time to perform changes, personnel, engineering effort, and risk [123, 509, 510]. Further information about the cost of evolving SMARTERCONTEXT at runtime are available in the evaluation section presented in [822].

As introduced in Section 8.2 engineering techniques applicable to self-adaptive software systems [177] can be used to evolve software systems dynamically. Nevertheless, runtime evolution is not always the best solution given the complexity added by the automation of evolution tasks. As an alternative to decide whether or not to implement runtime software evolution we envision the analysis of its benefit-cost ratio (BCR). We define the BCR of runtime software evolution as a function of the three variables mentioned above: frequency, uncertainty, and off-line evolution cost.

Figure 8.2 characterizes the application of runtime versus off-line software evolution according to the variables that affect the BCR function. Figure 8.2(a) concerns scenarios where the cost of off-line software evolution is high, whereas Figure 8.2(b) scenarios where this cost is low. In both tables rows represent the frequency of changes in requirements and environments, columns the level of uncertainty of these changes, and cells whether the recommendation is to apply runtime or off-line software evolution. Dark backgrounds indicate high levels of complexity added by the application of runtime evolution. We refer to each cell as $cell_{i,j}$ where i and j are the row and column indices that identify the cell.

According to Figure 8.2, runtime evolution is the preferred alternative when the cost of evolving the system off-line is high, as for example in the application of SMARTERCONTEXT to the monitoring of SLAs that may be renegotiated while the e-commerce platform is in production. When off-line evolution is affordable and the frequency of changes is high, both alternatives apply. Nevertheless, it is important to analyze the value added by runtime evolution versus its complexity (cf. $cell_{1,1}$ in Figure 8.2(b)). The complexity added by runtime evolution lies in the automation of activities such as the ones defined in the change mini-cycle process (cf. Figure 8.1). That is, the system must be instrumented with monitors to identify the need for evolution, analyzers and planners to correlate situations and define evolution actions dynamically, executors to modify the system, runtime validation and verification tasks to assure the operation of the system after evolution, and runtime modelling mechanisms to preserve the coherence between the running system and its design



		Column 1	Column 2
	High off-line evolution cost	High uncertainty	Low uncertainty
Row 1	High frequency	runE	runE
Row 2	Low frequency	runE offE	runE

(a)

		Column 1	Column 2
	Low off-line evolution cost	High uncertainty	Low uncertainty
Row 1	High frequency	offE runE	runE
Row 2	Low frequency	offE	offE

Fig. 8.2: A characterization of runtime versus off-line software evolution in light of frequency of changes in requirements and environments, uncertainty of these changes, and cost of off-line evolution. Dark backgrounds indicate high levels of complexity added by the application of runtime evolution.

artifacts. The complexity of the system is augmented since the functionalities that realize these tasks at runtime not only require qualified engineers for their implementation, but also become system artifacts that must be managed and evolved.

Under high levels of uncertainty runtime evolution is an alternative to be considered despite its complexity. This is because the variables that characterize the execution environment of the running system are unbound at design or development time, but bound at runtime. For example, in many cases it is infeasible to anticipate at design time or development time changes to be implemented in SMARTERCONTEXT. On the contrary, the runtime evolution scenarios that expose the lowest complexity are those with low uncertainty (cf. column 2 in both matrices). This is because the system operates in a less open environment where the evolution process can be characterized better at design and development time. As a result, the functionalities used to evolve the system at runtime are less affected by unforeseeable context situations. For example, it is possible to manually program runtime evolution functionalities to guarantee more demanding throughput levels during well known shopping seasons such as *Black Friday*.[1] When the off-line evolution cost is high and the uncertainty

[1] On Black Friday, the day after Thanksgiving Day in USA, sellers offer unbeatable deals to kick off the shopping season thus making it one of the most profitable days. Black Friday is

is low, runtime evolution seems to be the best alternative (cf. $cell_{1,2}$ and $cell_{2,2}$ in Figure 8.2(a)). Complexity is directly proportional to uncertainty. Therefore, when uncertainty and the cost of off-line evolution are low, runtime evolution is still a good option if the frequency of changes that require the evolution of the system are extremely high (cf. $cell_{1,2}$ in Figure 8.2(b)). In contrast, under low change frequencies, high levels of uncertainty and low off-line evolution costs, off-line evolution constitutes the best alternative (cf. $cell_{2,1}$ in Figure 8.2(b)). Moreover, off-line evolution is the best option when evolution cost, change frequencies and uncertainty are low (cf. $cell_{2,2}$ in Figure 8.2(b)).

Recalling the evolution scenario described in Section 8.2, it is possible to illustrate the application of the characterization of the BCR variables presented in Figure 8.2 to decide whether to implement runtime or off-line software evolution for adding new functionalities to the SmarterContext monitoring infrastructure. Regarding the cost of off-line software evolution, the implementation of manual maintenance tasks on the monitoring infrastructure of the e-commerce platform is clearly an expensive and complex alternative. First, the manual deployment and undeployment of components is expensive in terms of personnel. Second, the evolution time can be higher than the accepted levels, therefore the risk of violating contracts and losing customers increases. In particular, this is because renegotiations in SLAs cannot always be anticipated and must be addressed quickly. Uncertainty can also be high due to the variability of requirements among the retailers that use the e-commerce platform, thus increasing the frequency of changes in requirements. For example, new retailers may subscribe to the e-commerce platform with quality of service requirements for which the monitoring instrumentation is unsupported. Therefore, the system must evolve rapidly to satisfy the requisites of new customers. These high levels of uncertainty and frequency of changes in requirements require runtime support for monitoring, system adaptation, and assurance.

8.4 Dimensions of Runtime Software Evolution

Software evolution has been analyzed from several dimensions comprising, among others, the *what*, *why* and *how* dimensions of the evolution process [590]. The *what* and *why* perspectives focus on the software artifacts to be evolved (e.g., the software architecture of SmarterContext) and the reason for evolution (e.g., a new monitoring requirement), respectively. The *how* view focuses on the means to implement and control software evolution (e.g., using structural self-adaptation to augment the functionality of the system by deploying new software components at runtime). Figure 8.3 summarizes these perspectives as dimensions of *runtime* software evolution. With respect to the *why* perspective, the emphasis is on changing requirements (cf. Chapter 1), malfunctions, and changing execution environments as causes of run-

the day in which retailers earn enough profit to position them "in the black" – an accounting expression that refers to the practice of registering profits in black and losses in red. Source: http://www.investopedia.com.

time evolution. Regarding the *what*, the focus is on system goals, system structure and behavior, and design specifications as artifacts susceptible to runtime evolution. For example models and metamodels as studied in Chapter 2. Concerning the *how*, the attention is on methods, frameworks, technologies, and techniques gleaned from control theory and the engineering of self-adaptive software (SAS) systems. The elements highlighted in gray correspond the answers to the why, what and how questions for the case study presented in Section 8.2.

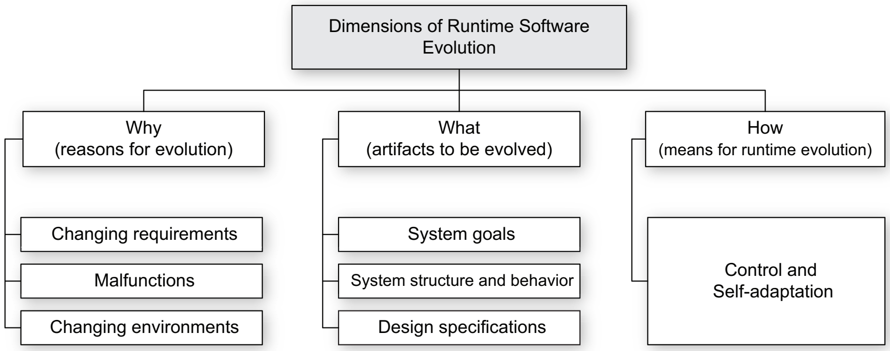

Dimensions of Runtime Software Evolution
Why (reasons for evolution)
What (artifacts to be evolved)
How (means for runtime evolution)
Changing requirements
Malfunctions
Changing environments
System goals
System structure and behavior
Design specifications
Control and Self-adaptation

Fig. 8.3: Characterizing dimensions of runtime software evolution. The highlighted elements relate to the SMARTERCONTEXT case study.

Software evolution has been characterized through the *laws of software evolution* proposed by Lehman [509, 553]. Lehman's laws apply to *E-type* systems, the term he used to refer to software systems that operate in real world domains that are potentially unbounded and susceptible to continuing changes [507]. Therefore, it is clear that when Lehman proposed his laws of software evolution back in the seventies, he focused on systems that operate in real world situations and therefore are affected by continuing changes in their operation environments [509, 553]. Hence, Lehman's laws corroborate the need for preserving the qualities of software systems in changing environments, which may require the implementation of runtime evolution capabilities depending on the cost of off-line evolution, the uncertainty of the environment and the frequency of changes. Lehman's laws of software evolution can be summarized as follows:

1. *Continuing change.* E-type systems must adapt continuously to satisfy their requirements.
2. *Increasing complexity.* The complexity of E-type systems increases as a result of their evolution.
3. *Self-regulation.* E-type system evolution processes are self-regulating.
4. *Conservation of organizational stability.* Unless feedback mechanisms are appropriately adjusted in the evolution process, the average effective rate in an evolving E-type system tends to remain constant over the product lifetime.

5. *Conservation of familiarity.* The incremental growth and long term growth rate of E-type systems tend to decline.
6. *Continuing growth.* The functional capability of E-type systems must be continually increased to maintain user satisfaction over the system lifetime.
7. *Declining quality.* Unless adapted according to changes in the operational environment, the quality of E-type systems decline.
8. *Feedback system.* E-type evolution processes are multi-level, multi-loop, multi-agent feedback systems [553].

Considering the *continuing change* law, highly dynamic software systems such as SMARTERCONTEXT need not only adapt continuously to satisfy their requirements, but in many cases do it at runtime. One of the key benefits of evolving a system at runtime is to be able to verify assumptions made at design time. Although valid when the system was designed, some of these assumptions become invalid over time. For example, the QoS requirements of the e-commerce company in the application scenario described in Section 8.2 may vary over time. Moreover, new monitoring requirements must be addressed to satisfy changes in SLAs. To counteract the effects of this first law, traditional software evolution focuses on the off-line activities defined in the change mini-cycle process (cf. Figure 8.1). On the contrary, runtime evolution argues for the instrumentation of software systems with feedback-control capabilities that allow them to manage uncertainty and adapt at runtime, by performing these tasks while the system executes. Of course, due to the *increasing complexity* law, the trade-off between runtime and off-line evolution affects the level of automation and the instrumentation required to evolve the system without risking its operation. The complexity of highly dynamic software systems includes aspects such as the monitoring of the execution environment, the specification and management of changing system requirements, the implementation of dynamic mechanisms to adjust the software architecture and business logic, the preservation of the coherence and causal connection among requirements, specifications and implementations, and the validation of changes at runtime.

The *self-regulation* law stated by Lehman characterizes software evolution as a self-regulating process. In highly dynamic software systems self-regulation capabilities are important across the software life cycle, in particular at runtime. One implication, among others, is the capability of the system to decide when to perform maintenance and evolution tasks and to assure the evolution process. Self-regulating software evolution can be realized by enabling software systems with feedback control mechanisms [147, 212, 553, 622].

Laws *continuing growth* and *declining quality* are undeniably connected with the capabilities of a software system to evolve at runtime. Continuing growth argues for the need of continually increasing the functionalities of the system to maintain user satisfaction over time. Similarly, declining quality refers to the need for continually adapting the system to maintain the desired quality properties.

Regarding declining quality, runtime software evolution can effectively deal with quality attributes whose preservation depends on context situations [820, 899].

8.5 Control in Runtime Software Evolution

Control is an enabling technology for software evolution. At runtime, control mechanisms can be realized using self-adaptation techniques [147]. *Feedback control* concerns the management of the behavior of dynamic systems. In particular, it can be applied to automate the control of computing and software systems [147, 388]. From the perspective of software evolution, feedback control can be defined as the use of algorithms and feedback for implementing maintenance and evolution tasks [553, 625].

8.5.1 Feedback Control

Feedback control is a powerful tool for uncertainty management. As a result, self-evolving software systems based on feedback loops are better prepared to deal with uncertain evolution scenarios. This is the reason why the BCR of runtime evolution is higher under uncertain environments (cf. Section 8.3). Uncertainty management using feedback control is realized by monitoring the operation and environment of the system, comparing the observed variables against reference values, and adjusting the system behavior accordingly. The goal is to adapt the system to counteract disturbances that can affect the accomplishment of its goals.

8.5.2 Feedforward Control

Feedforward control, which operates in an open loop, can also benefit the evolution of software systems. The fundamental difference between feedback and feedforward control is that in the first one control actions are based on the deviation of measured outputs with respect to reference inputs, while in the latter control actions are based on plans that are fed into the system. These plans correspond to actions that are not associated with uncertain changes in the execution environment. Another application of feedforward control is the empowerment of business and system administrators to modify system goals (e.g., non-functional requirements) and policies that drive the functionalities of feedback controllers. Feedforward control could be exploited using policies and pre-defined plans as the mechanism to control runtime software evolution tasks. Therefore, feedforward control provides the means to manage short-term evolution according to the long-term goals defined in the general evolution process.

The application of feedback and feedforward control to runtime software evolution can be analyzed in light of the dimensions of software evolution depicted in Figure 8.3. With respect to the *why* dimension, feedback control applies to runtime evolution motivated by malfunctions or changing environments, and feedforward to runtime evolution originated in changing requirements. The reasons for this distinction are that in the first case the symptoms that indicate the need for evolution result from the monitoring of the system and its environment and the analysis of the observed conditions. In the second case, the evolution stimulus comes from the direct action of an external entity (e.g., a system administrator). For example, in the case of SMARTERCONTEXT feedback control allows the monitoring infrastructure to replace failing sensors automatically, whereas feedforward enables SMARTER-CONTEXT to deploy new monitoring artifacts or modify the monitoring logic to address changes in SLAs. With respect to the *what* dimension, pure control-based evolution techniques better apply to the modification of the system behavior or its computing infrastructure (this does not involve the software architecture). The reason is that pure control-based adaptation actions are usually continuous or discrete signals calculated through a mathematical model defined in the controller [899].

The feedback loop, a foundational mechanism in control theory, is a reference model in the engineering of SAS systems [622, 772]. Therefore, the implementation of runtime evolution mechanisms based on feedback control and self-adaptation requires the understanding of the feedback loop, its components, and the relationships among them. Figure 8.4 depicts the feedback loop model from control theory. To explain its components and their relationships, we will extend our runtime software evolution case study described in Section 8.2. These extensions are based on examples written by Hellerstein et al. [388].

Imagine that the service-oriented e-commerce system that is monitored by the SMARTERCONTEXT monitoring infrastructure provides the online retailing platform for several companies worldwide. Important concerns in the operation of this system are its continuous availability and its operational costs. The company that provides this e-commerce platform is interested in maximizing system efficiency and guaranteeing application availability, according to the needs of its customers. Concerning efficiency, the company defines a performance quality requirement for the service that processes purchase orders. The goal is to maximize the number of purchase orders processed per time unit. Regarding availability, the business implements a strategy based on redundant servers. The objective is to guarantee that the operation of the e-commerce platform upon eventual failures in its infrastructure. After the system has been in production for a certain time period, the company realizes that the system capacity to process purchase orders is insufficient to satisfy the demand generated by *Black Friday* and by different special offers placed at strategic points during the year. To solve this problem, the company may decide to extend its physical infrastructure capacity to guarantee availability under peak load levels. Nevertheless, this decision will affect efficiency and costs since an important part of the resources will remain unused for long periods of time, when the system load is at its normal levels. To improve the use of resources, the company may decide to perform periodic maintenance tasks manually to increase or decrease system capacity according to the demand. However, this strategy not only increases the costs and complexity of maintaining and evolving the system, but is also ineffective since changes in the demand cannot always be anticipated and may arise quickly (e.g., in intervals of minutes when a special discount becomes popular in a social network) and frequently. Similarly, the strategy based on redundant servers to guarantee availability may challenge the maintenance and evolution of the system when performed manually. First, the use of redundant servers to address failures in the infrastructure must not compromise the contracted capacity of the system. Thus, servers must be replaced only with machines of similar specifications. Second, this strategy must take into account the capacity levels contracted with each retailer, which may vary not only over time, but also from one retailer to another.

The target system represents the system to be evolved dynamically using self-adaptation (e.g., our e-commerce platform). System requirements correspond to *reference inputs* (label (A) in Figure 8.4). Suppose that for the e-commerce company to guarantee the availability of its platform, it implements the redundant server strategy by controlling the CPU utilization of each machine. For this, it must maintain servers working below their maximum capacity in such a way that when a server fails, its load can be satisfied by the others. The reference input corresponds to the desired CPU utilization of the servers being controlled. The system is monitored continuously by comparing the actual CPU usage, the *measured output* (label (B)), against the reference input. The difference between the measured output and the reference input is the *control error* (label (C)). The controller implements a mathematical function (i.e., transfer function) that calculates the *control input* (label (D)) based on the error. Control inputs are the stimuli used to affect the behavior of the target system. To control the desired CPU usage, the control input is defined as the maximum number of connections that each controlled server must satisfy. The measured output can also be affected by external *disturbances* (label (E)), or even by the *noise* (label (F)) caused by the system evolution. *Transducers* (label (G)) translate the signals coming from sensors, as required by the comparison element (label (H), for example to unify units of measurement). In this scenario the *why* dimension of runtime software evolution corresponds to malfunctions, the *what* to the processing infrastructure, and the *how* to self-adaptation based on feedback control.

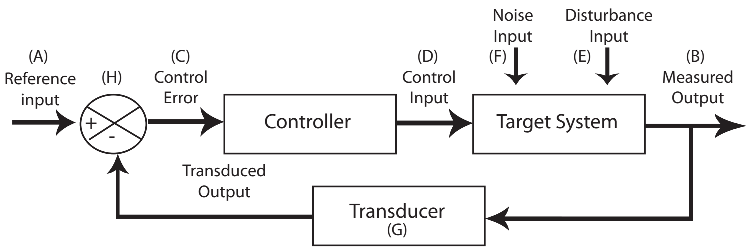


Fig. 8.4: Classical block diagram of a feedback control system [388]. Short-term software evolution can be realized through feedback loops that control the behavior of the system at runtime.

To Probe Further

Readers interested in studying the application of feedback loops to the engineering of self-adaptation mechanisms for software systems may refer to [147, 388, 622, 772, 900].

8.5.3 Adaptive Control

From the perspective of control theory, *adaptive control* concerns the automatic adjustment of control mechanisms. Adaptive control researchers investigate parameter adjustment algorithms that allow the adaptation of the control mechanisms while guaranteeing global stability and convergence [268]. Control theory offers several reference models for realizing adaptive control. We focus our attention on two of them, *model reference adaptive control* (MRAC) and *model identification adaptive control* (MIAC).

8.5.3.1 Model Reference Adaptive Control (MRAC)

MRAC, also known as *model reference adaptive system* (MRAS), is used to implement controllers that support the modification of parameters online to adjust the way the target system is affected (cf. Figure 8.5). A reference model, specified in advance, defines the way the controller's parameters affect the target system to obtain the desired output. Parameters are adjusted by the adaptation algorithm based on the control error, which is the difference of the measured output and the expected result according to the model.

In runtime software evolution, MRAC could applied to the modification of the evolution mechanism at runtime. Since the controller uses the parameters received from the adaptation algorithm to evolve the target system, the control actions implemented by the controller could be adjusted dynamically by modifying the reference model used by the adaptation algorithm. The application of MRAC clearly improves the dynamic nature of the evolution mechanism, which is important for scenarios where the *why* dimension focuses on changes in requirements. Nevertheless, MRAC has a limited application in scenarios with high levels of uncertainty because it is impractical to predict changes in the reference model.

8.5.3.2 Model Identification Adaptive Control (MIAC)

In MIAC, the reference model that allows parameter estimation is identified or inferred at runtime using system identification methods. As depicted in Figure 8.6, the

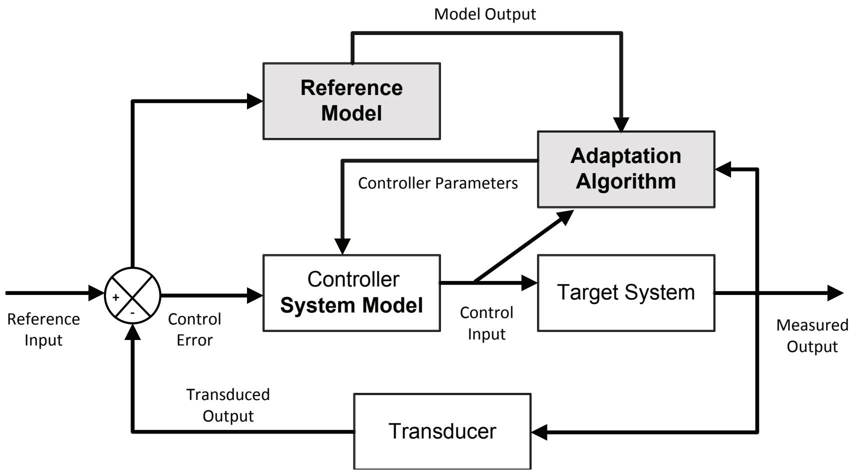


Fig. 8.5: Model Reference Adaptive Control (MRAC)

control input and measured output are used to identify the reference model (system identification). Then, the new model parameters are calculated and sent to the adjustment mechanism which calculate the parameters that will modify the controller.

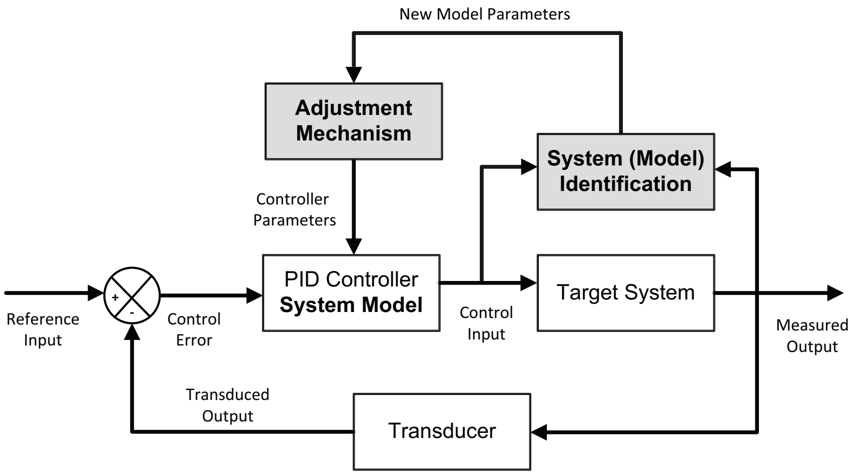


Fig. 8.6: Model Identification Adaptive Control (MIAC)

In the context of runtime software evolution, MIAC could support the detection of situations in which the current evolution strategy is no longer effective. Moreover, it could be possible to exploit MIAC to adjust the evolution mechanism fully or semi automatically. Since the reference model used to realize the controller's actions is synthesized from the system, MIAC seems more suitable for highly uncertain scenarios where the *why* dimension of software evolution corresponds to changes in the environment.

8.6 Self-Adaptive Software Systems

Another dimension that has been used to characterize software evolution refers to the *types of changes* to be performed in the evolution process [590], for which several classifications have been proposed [170, 813]. In particular, the ISO/IEC standard for software maintenance proposes the following familiar classification: *adaptive maintenance*, defined as the modification of a software product after its delivery to keep it usable under changing environmental conditions; *corrective maintenance*, as the reactive modification of the system to correct faults; and *perfective maintenance*, as the modification of the software to improve its quality attributes [424]. These maintenance types can be implemented using different self-adaptation approaches [739]. For example, adaptive maintenance can be realized through context-aware self-reconfiguration, corrective maintenance through self-healing, and perfective maintenance through self-optimization. Self-adaptive software (SAS) systems are software applications designed to adjust themselves, at runtime, with the goal of satisfying requirements that either change while the system executes or depend on changing environmental conditions. For this, SAS systems are usually instrumented with a feedback mechanism that monitors changes in their environment—including their own health and their requirements—to assess the need for adaptation [177, 221]. In addition to the monitoring component, the feedback mechanism includes components to analyze the problem, decide on a plan to remedy the problem, effect the change, as well as validate and verify the new system state. This feedback mechanism, also called adaptation process, is similar to the continuous feedback process of software evolution characterized by the change mini-cycle model (cf. Figure 8.1).

As analyzed in Section 8.3, under highly changing requirements and/or execution environments, it is desirable to perform evolution activities while the system executes. In particular when the off-line evolution is expensive. By instrumenting software systems with control capabilities supported by self-adaptation, it is possible to manage short-term evolution effectively. Indeed, the activities performed in the adaptation process can be mapped to the general activities of software evolution depicted by the change mini-cycle model. Therefore, self-adaptation can be seen as a short-term evolution process that is realized at runtime. SAS system techniques can greatly benefit the evolution of highly dynamic software systems such in the case of the SMARTERCONTEXT monitoring infrastructure of our case study. The monitoring requirements addressed by SMARTERCONTEXT are highly changing since they depend on contracted conditions that may be modified after the e-commerce system is in production. Therefore, to preserve the relevance of monitoring functionalities with respect to the conditions specified in SLAs, SMARTERCONTEXT relies on self-adaptation to address new functional requirements by evolving the monitoring infrastructure at runtime. This section introduces the engineering foundations of SAS systems and illustrates their application to runtime software evolution.

To Probe Further

The survey article by Salehie and Tahvildari presents an excellent introduction to the state of the art of SAS systems [739]. Their survey presents a taxonomy, based on *how*, *what*, *when* and *where* to adapt software systems, as well as an overview of application areas and selected research projects. For research roadmaps on SAS systems please refer to [177, 221].

8.6.1 Self-Managing Systems

Self-managing systems are systems instrumented with self-adaptive capabilities to manage (e.g., maintain or evolve) themselves given high-level policies from administrators. and with minimal human intervention [457]. *Autonomic computing*, an IBM initiative, aims at implementing self-managing systems able to anticipate changes in their requirements and environment, and accommodate themselves accordingly, to address system goals defined by policies [457]. The ultimate goal of autonomic computing is to improve the efficiency of system operation, maintenance, and evolution by instrumenting systems with autonomic elements that enable them with self-management capabilities. Systems with self-management capabilities expose at least one of the four self-management properties targeted by autonomic computing: *self-configuration, self-optimization, self-healing,* and *self-protection.* This subsection presents concepts from autonomic computing and self-managing systems that are relevant to the evolution of highly dynamic software systems.

8.6.2 The Autonomic Manager

The *autonomic manager*, illustrated in Figure 8.7, is the fundamental building block of self-managing systems in autonomic computing. The autonomic manager can be used to realize runtime evolution. For this, it implements an intelligent control loop (cf. Figure 8.4) that is known as the *MAPE-K* loop because of the name of its elements: the *monitor*, *analyzer*, *planner*, *executor*, and *knowledge base*. *Monitors* collect, aggregate and filter information from the environment and the target system (i.e., the system to be evolved), and send this information in the form of symptoms to the next element in the loop. *Analyzers* correlate the symptoms received from monitors to decide about the need for adapting the system. Based on business policies, *planners* define the maintenance activities to be executed to adapt or evolve the system. *Executors* implement the set of activities defined by planners. The knowledge base enables the information flow along the loop, and provides persistence for historical information and policies required to correlate complex situations.

The autonomic manager implements *sensors* and *effectors* as manageability endpoints that expose the state and control operations of managed elements in the system (e.g., the service that processes purchase orders and the managed servers in our e-commerce scenario). Sensors allow autonomic managers to gather information from both the environment and other autonomic managers. Effectors have a twofold function. First they provide the means to feed autonomic managers with business policies that drive the adaptation and evolution of the system. Second they provide the interfaces to implement the control actions that evolve the managed element. Managed elements can be either system components or other autonomic managers.

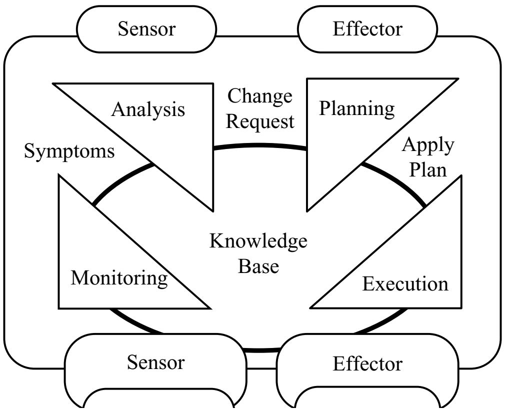


Fig. 8.7: The autonomic manager [457]. Each autonomic manager implements a feedback loop to *monitor* environmental situations that may trigger the adaptation of the system, *analyze* the need for adapting, *plan* the adaptation activities, and *execute* the adaptation plan.

To Probe Further

One of the most important articles on the notion of an autonomic manager and its applications is the 2006 IBM Technical Report entitled "An Architectural Blueprint for Autonomic Computing (Fourth Edition)" [417]. In another article, Dobson et al. argue for the integration of autonomic computing and communications and thus surveyed the state-of-the-art in autonomic communications from different perspectives [258].

Figure 8.8 depicts the mapping (cf. dashed arrows) between phases of the software evolution process defined by the change mini-cycle model (cf. white rounded boxes in the upper part of the figure) and the phases of the MAPE-K loop implemented by the autonomic manager (cf. gray rounded boxes at the bottom of the figure). On the one hand, the change mini-cycle model characterizes the activities of the "traditional" long-term software evolution process which is performed off-line [936]. On the other hand, the phases of the MAPE-K loop are realized at runtime. Therefore, autonomic managers can be used to realize short-term software evolution at runtime. The last two activities of the change mini-cycle model have no mapping to the MAPE-K loop process. However, this does not mean that these activities are not addressed in the implementation of self-adaptation mechanisms for software systems. Validation and verification (V&V) refer to the implementation of assurance mechanisms that allow the certification of the system after its evolution. Re-documentation concerns the preservation of the relevance of design artifacts with respect to the new state of the evolved system. To realize runtime evolution through self-management capabilities of software systems these activities must also be performed at runtime. Indeed, these are challenging topics subject of several research initiatives in the area of software engineering for self-adaptive and self-managing software systems. In particular, assurance concerns can be addressed through the implementation of V&V tasks along the adaptation feedback loop [823]. The preservation of the relevance between design artifacts and the evolving system can be addressed through the implementation of runtime models [92]. Runtime V&V and runtime models are foundational elements of SAS systems also addressed in this chapter.

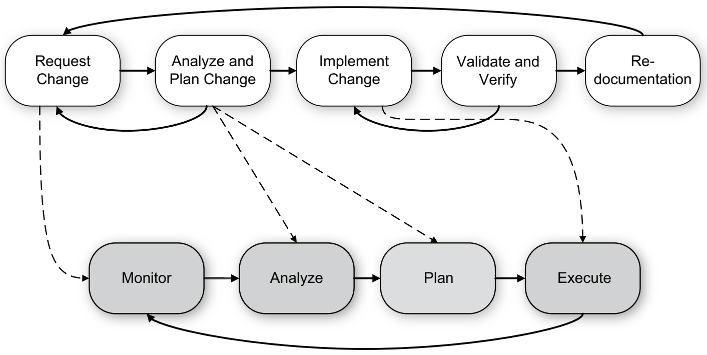


Fig. 8.8: Mapping the phases of the software evolution change mini-cycle (cf. Figure 8.1) to the phases of the MAPE-K loop implemented by the autonomic manager (cf. Figure 8.7)

8.6.3 The Autonomic Computing Reference Architecture

The autonomic computing reference architecture (ACRA), depicted in Figure 8.9, provides a reference architecture as a guide to organize and orchestrate self-evolving (i.e., autonomic) systems using autonomic managers. Autonomic systems based on ACRA are defined as a set of hierarchically structured building blocks composed of orchestrating managers, resource managers, and managed resources. Using ACRA, software evolution policies can be implemented as resource management policies into layers where system administrator (manual manager) policies control lower level policies. System operators have access to all ACRA levels.

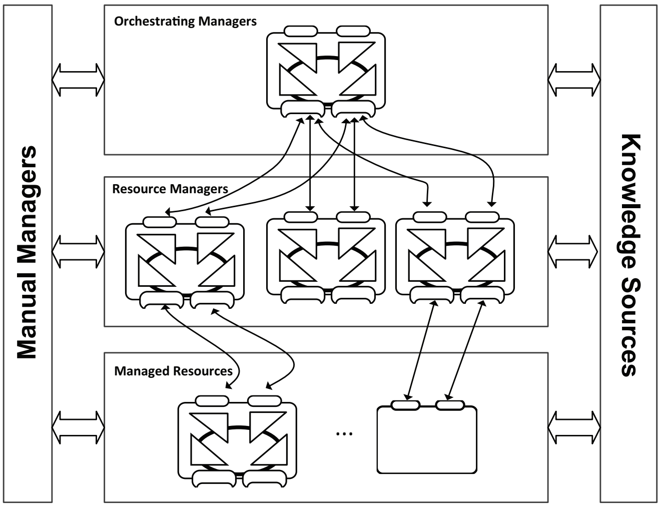


Fig. 8.9: The Autonomic Computing Reference Architecture (ACRA) [417]

To Probe Further

Over the past thirty years, researchers from different fields have proposed many three-layer models for dynamic systems. In particular, researchers from control engineering, AI, robotics, software engineering, and service-oriented systems have devised—to a certain extent independently—reference architectures for designing and implementing control systems, mobile robots, autonomic systems, and self-adaptive systems. Seminal three-layer reference architectures for self-management are the hierarchical intelligent control system (HICS) [745], the concept of adaptive control architectures [48], Brooks' layers of competence [145], Gats Atlantis architecture [317], IBM's ACRA—autonomic computing reference architecture [417, 457], and Kramer and Magee's self-management architecture [478, 479]. Oreizy et al. introduced the concept of *dynamic architecture evolution* with their the *Figure 8* model separating the concerns of adaptation management and evolution management [668–670]. Dynamico is the most recent three-layer reference model for context-driven self-adaptive systems [900]. The key idea in these layered architectures is to build levels of control or competence. The lowest level controls individual resources (e.g., manage a disk). The middle layer aims to achieve particular goals working on individual goals concurrently (e.g., self-optimizing or self-healing). The highest level orchestrates competing or conflicting goals and aims to achieve overall system goals (e.g., minimizing costs).

ACRA is useful as a reference model to design and implement runtime evolution solutions based on autonomic managers. For example, as depicted in Figure 8.10, the ACRA model can be used to derive architectures that orchestrate the interactions among managers, deployed at different levels, to control the evolution of goals, models, and systems. Recall from Figure 8.3 that goals, models, and the structure and behavior of systems correspond to the artifacts that characterize the *what* dimension of runtime software evolution (i.e., the elements of the system susceptible to dynamic evolution). Runtime evolution mechanisms based on ACRA can take advantage of both feedforward and feedback control. Feedforward control can be implemented through the interactions between manual managers (i.e., business and system administrators) and the autonomic managers deployed at each level of the runtime evolution architecture (cf. dashed arrows in Figure 8.10). Feedback control can be exploited to orchestrate the evolution of related artifacts located at two different levels (i.e., inter-level feedback represented by continuous arrows), and to implement autonomic managers at each level of the architecture (i.e., intra-level feedback control represented by autonomic managers). Thick arrows depict the information flow between knowledge bases and autonomic managers.

The evolution architecture depicted in Figure 8.10 applies to our e-commerce scenario (cf. Section 8.2). Goals correspond to QoS requirements that must be satisfied for the retailers that use the e-commerce platform (e.g., performance as a measure of efficiency). Models correspond to specifications of the software architecture configurations defined for each QoS requirement. System artifacts correspond to the actual servers, components, and services. As explained in Section 8.5, feedback control can be exploited to evolve the configuration of servers with the goal of guaranteeing system availability. For this, autonomic managers working at the top level of the architecture monitor and manage changes in system goals. Whenever a goal is modified, these managers use inter-level control to trigger the evolution of models at the next level. Subsequently, autonomic managers in charge of controlling the evolution of models provide managers at the bottom level with control actions that will trigger the reconfiguration of the system. In this application scenario feedforward enables the runtime evolution of the system to satisfy changes in requirements of retailers. For example, when a new retailer becomes a customer, business administrators can feed the evolution mechanism with new goals to be satisfied. The definition of a new goal triggers the inter- and intra-level feedback mechanisms to evolve models and systems according to the new requirement. When the new requirement cannot be satisfied with existing models and adaptation strategies, feedforward allows system administrators to feed the system with new ones.

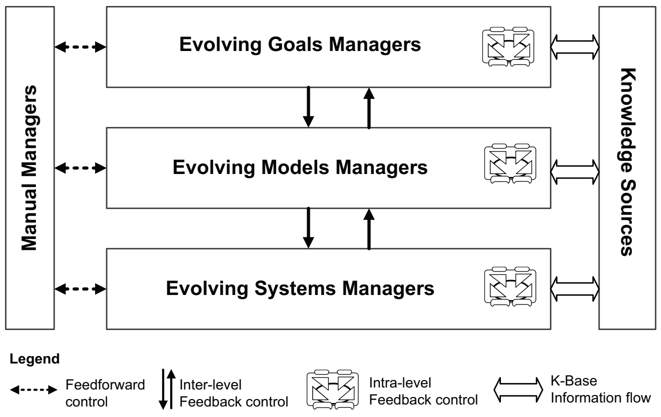


Fig. 8.10: A runtime evolution architecture based on ACRA supporting the evolution of software artifacts at the three levels of the *what* dimension: goals, models, and system structure and behavior. Dashed arrows represent feedforward control mechanisms, continuous arrows inter-level feedback control mechanisms, autonomic managers intra-level feedback control, and thick arrows the information flow between autonomic managers and knowledge bases.

8.6.4 Self-Management Properties

In autonomic computing self-managing systems expose one or more of the following properties: self-configuration, self-optimization, and self-protection, self-healing [457]. These properties are also referred to as the *self-* adaptation properties* of SAS systems and concern the *how* perspective of the runtime software evolution dimensions depicted in Figure 8.3. Of course, they also relate to the *why* and *what* perspectives since runtime evolution methods target evolution goals and affect artifacts of the system. Self-* properties allow the realization of maintenance tasks at runtime. For example, self-configuration can be used to realize adaptive maintenance, which goal is to modify the system to guarantee its operation under changing environments; self-healing and self-protection to realize corrective maintenance, which goal is to recover the system from failures or protect it from hazardous situations; and self-optimization to implement perfective maintenance, which goal is usually to improve or preserve quality attributes. Each self-* property can be further sub-classified. For example self-recovery is often classified as a self-healing property.

Kephart and Chess characterized self-* properties in the context of autonomic computing [457]. In this subsection we characterize these properties in the context of runtime software evolution. Table 8.1 presents selected examples of self-adaptive solutions characterized with respect to the three dimensions of runtime software evolution depicted in Figure 8.3, where the *how* dimension corresponds to self-* properties.

Table 8.1: Examples of self-adaptation approaches characterized with respect to the *why*, *what* and *how* dimensions of runtime software evolution.

Approach	Why	What	How
Cardellini et al. [166]		System structure	
Dowling and Cahill [261]	Changing environments	System structure	
Parekh et al. [678]		System behavior	Self-configuration
Tamura et al. [821]		System structure	
Villegas et al. [898]	Changing requirements	Structure/behavior	
Kumar et al. [484]			
Solomon et al. [786]	Changing environments	System structure	Self-optimization
Appleby et al. [44]			
Baresi and Guinea [73]		System behavior	
Candea et al. [156]	Malfunctions	System structure	Self-healing
Ehrig et al. [270]		System behavior	
White et al. [917]	Malfunctions	System behavior	Self-protection

Self-Configuration

Self-configuration is a generic property that can be implemented to realize any other self-* property. Systems with self-configuration capabilities reconfigure themselves, automatically, based on high level policies that specify evolution goals, as well as reconfiguration symptoms and strategies. Self-configuring strategies can affect either the system's behavior or structure. Behavioral reconfiguration can be achieved by modifying parameters of the managed system, whereas structural reconfiguration can be achieved by modifying the architecture. Self-configuration strategies can be applied to our exemplar e-commerce system to guarantee new quality attributes that may be contracted with retailers. This could be realized by reconfiguring the system architecture at runtime based on design patterns that benefit the contracted qualities [820]. In self-configuration approaches the *why* dimension of runtime software evolution may correspond to changing requirements or environments, and malfunctions, whereas the *what* dimension concerns the system structure or behavior, depending on the way the target system is adapted.

The first group of approaches in Table 8.1 illustrates the application of self-configuration as the means to runtime software evolution. The prevalent reason for evolution (i.e., the *why* dimension) is changing environments, except the approach by Villegas et al. whose reason for evolution is the need for supporting changes in requirements. Cardellini and her colleagues implemented Moses (MOdel-based SElf-Adaptation of SOA systems), a self-configuration solution for service selection and workflow restructuring, with the goal of preserving selected quality attributes under environmental disruptions [166]. To reconfigure the system, MOSES's self-adaptive solution, based on a runtime model of the system, calculates the service composition that better satisfies the QoS levels contracted with all the users of the system. This mechanism is based on a linear programming optimization problem that allows them to efficiently cope with changes in the observed variables of the execution environment. Downling and Cahill [261], as well as Tamura et al. [820, 821] proposed self-configuration approaches applied to component-based software systems with the goal of dealing with the violation of contracted qualities due to changes in the environment. Both approaches use architectural reflection to affect the system structure. Self-adaptation approaches can be classified along a spectrum of techniques that ranges from pure control-based to software-architecture-based solutions [899]. Control-based approaches are often applied to the control of the target system's behavior and hardware infrastructure rather than its software architecture. Parekh et al. proposed a more control theory oriented approach for self-configuration [678], where the *what* dimension of software evolution concerns the system behavior. The evolution objective in this case is also the preservation of quality properties, and the evolution is realized through a controller that manipulates the target system's tuning parameters. Villegas et al. implemented SMARTERCONTEXT, a context management infrastructure that is able to evolve itself to address changes in monitoring requirements [894, 898]. SMARTERCONTEXT can support dynamic monitoring in several self-adaptation and runtime evolution scenarios. In particular, they have applied their solution to support the runtime evolution of software systems with the goal of addressing changes in SLA dynamically.

Self-Optimization

Perfective maintenance often refers to the improvement of non-functional properties (i.e., quality requirements and *ility* properties) of the software system such as performance, efficiency, and maintainability [170]. Perfective maintenance can be realized at runtime using self-optimization. Self-optimizing systems adapt themselves to improve non-functional properties according to business goals and changing environmental situations. For example, in our e-commerce scenario the capacity of the system can be improved through a self-configuration mechanism implemented to increase the number of services available for processing purchase orders. This operation affects the software architecture of the system, and is performed to address changing context situations (e.g., critical changes in the system load due to shopping seasons or special offers that become extremely popular). Therefore, the *what* dimension concerns the structure of the system, and the *why* dimension changing environments that must be monitored continuously.

Examples of self-optimization mechanisms implemented through self-adaptation are the Océno approach proposed by Appleby et al. [44], the middleware for data stream management contributed by Kumar et al. [484], and the self-adaptation approach for business process optimization proposed by Solomon et al. [786]. Appleby and her IBM colleagues proposed a self-optimization approach that evolves the infrastructure of an e-business computing utility to satisfy SLAs under peak-load situations. In their approach, self-adaptive capabilities support the dynamic allocation of computing resources according to metrics based on the following variables: active connections/server, overall response time, output bandwidth, database response time, throttle rate, admission rate, and active servers. These variables constitute the relevant context that is monitored to decide whether or not to evolve the system. Kumar and colleagues proposed a middleware that exposes self-optimizing capabilities, based on overlay networks, to aggregate data streams in large-scale enterprise applications. Their approach deploys a data-flow graph as a network overlay over the enterprise nodes. In this way, processing workload is distributed thus reducing the communication overhead involved in transmitting data updates. The configuration and optimization of the deployed overlay is performed by an autonomic module that focuses on maximizing a utility function where monitoring conditions involve several internal context variables. Solomon et al. proposed an autonomic approach that evolves software systems to optimize business processes. At runtime, their approach predicts the business process that better addresses SLAs while optimizing system resources. The prediction is based on a simulation model whose parameters are tuned at runtime by tracking the system with a particle filter.

Balasubramanian et al. introduce a mathematical framework that adds structure to problems in the realm of self-optimization for different types of policies [66]. Structure is added either to the objective function or the constraints of the optimization problem to progressively increase the quality of the solutions obtained using the greedy optimization technique. They characterized and analyzed several optimization problems encountered in the area of self-adaptive and self-managing systems to provide quality guarantees for their solutions.

Self-Healing

Corrective maintenance is a main issue in software evolution. In complex systems it is particularly challenging due to the difficulty of finding the root cause of failures. This is in part because the situation that triggers the failure may be no longer available by the time the analysis is performed. Self-healing systems are equipped with feedback loops and runtime monitoring instrumentation to detect, diagnose, and fix malfunctions that are originated from the software or hardware infrastructure. In our e-commerce runtime evolution example, self-healing applies to the assurance of the system availability. In particular, the runtime evolution strategy described in Section 8.5 applies behavioral reconfiguration to control the maximum number of connections that each server can satisfy, with the goal of having always processing capacity available to replace faulty servers. In this situation, the *what* dimension of runtime evolution corresponds to the behavior of the system, whereas the *why* corresponds to malfunctions. The third group in Table 8.1 corresponds to runtime evolution approaches focused on self-healing. Self-healing evolution is usually triggered by malfunctions, and can affect both the structure (e.g., Candea et al. [156]) and behavior of the system (e.g., Baresi and Guinea [73], as well as Ehrigh et al. [270]). Candea et al. built a self-recovery approach that uses recursive micro-reboots to optimize the maintenance of Mercury, a commercial satellite ground station that is based on an Internet service platform. Baresi and Guinea implemented a self-supervising approach that monitors and recovers service-oriented systems based on user-defined rules. Recovery strategies in their approach comprise a set of atomic recovery actions such as the rebinding and halt of services. Ehrigh et al. implemented self-healing capabilities using typed graphs and graph-based transformations to model the system to be adapted and the adaptation actions, respectively.

Self-Protection

Self-protection can be classified as perfective maintenance. Self-protected systems are implemented with feedback loops to evolve themselves at runtime to counteract malicious attacks and prevent failures. White et al. proposed an approach and methodology to design self-protecting systems by exploiting model-driven engineering [917].

8.7 Self-Adaptation Enablers for Runtime Evolution

The continuing evolution of software systems, and the uncertain nature of execution environments and requirements have contributed to blur the line between design time (or development time) and runtime [72, 280]. As a result, activities that have been traditionally performed off-line must now also be performed at runtime. These activities include the maintenance of requirements and design artifacts, as well as the validation and verification (V&V) of software system. This section summarizes self-adaptation enablers from the perspective of runtime software evolution. We concentrate on requirements and models at runtime as well as runtime monitoring and V&V.

8.7.1 Requirements at Runtime

The concept *requirements at runtime* refers to the specification of functional and non-functional requirements using machine-processable mechanisms [748]. As discussed in Chapter 1 requirements form part of the evolving artifacts in a software system. Therefore, having requirement specifications available at runtime is particularly important for evolution scenarios where the *why* dimension corresponds to changing requirements, and the *what* to system goals. Aspects of runtime software evolution that rely on runtime specifications of requirements include:

- The specification of evolution goals and policies by business and system administrators.
- The control of evolving goals. For example, by the autonomic managers defined at the first level of runtime software evolution in the ACRA-based architecture depicted in Figure 8.10.
- The specification of adaptation properties that must be satisfied by evolution controllers. Adaptation properties refer to qualities that are important for controllers to operate reliably and without compromising the quality of the evolving system. Examples of adaptation properties include properties gleaned from control engineering such as short settling time, accuracy, small overshoot, and stability. A characterization of these properties from the perspective of software systems is available in [899].
- The monitoring of the execution environment and the system to identify the need for evolution.
- The preservation of the context-awareness along the evolution process. Runtime requirement specifications must maintain an explicit mapping with monitoring strategies to the adaptation of monitoring infrastructures as required by the evolution of the system.
- The management of uncertainty due to changes in requirements. Requirements are susceptible to changes in the environment, user preferences, and business goals. Thus, runtime specifications of requirements allow the system to maintain an explicit mapping between requirements and aspects that may affect them [921]. Moreover, they are crucial for the *re-documentation* phase of the software evolution process (cf. Figure 8.8), since they ease the maintenance of the coherence between the actual system implementation and its documentation.
- The validation and verification of the system along the runtime evolution process (cf. Figure 8.8). Requirements at runtime allow the specification of aspects to validate and verify [823].

8.7.2 Models at Runtime

The concept *models at runtime* refers to representations of aspects of the system that are specified in a machine-readable way, and are accessible by the system at runtime. In the context of runtime software evolution, runtime models provide the system with up-to-date information about itself and its environment. Moreover, runtime models are themselves artifacts that evolve with the system (i.e., design specifications as defined in the *what* dimension of software evolution, cf. Figure 8.3).

To Probe Further

The *models@run.time* research community defines a runtime model as a “causally connected self-representation of the associated system that emphasizes the structure, behavior, or goals of the system from a problem space perspective” [52, 92, 115]. In the context of self-adaptation and runtime evolution, the environment that affects the system in the accomplishment of its goals is also an aspect that requires runtime models for its specification [177, 221, 823, 900].

Runtime models provide effective means to evolve software systems at runtime. For example, in our e-commerce scenario administrators can modify the runtime model that specifies the software architecture to be implemented to improve the capacity of the system for processing purchase orders. The modification of the software architecture model will trigger the adaptation of the software system. This model-based evolution mechanism can be implemented using ACRA as depicted in Figure 8.10. The middle layer contains the autonomic managers in charge of evolving runtime models.

Runtime models also support the implementation of runtime evolution mechanisms based on adaptive control. In particular using MRAC and MIAC. In MRAC, the reference model used by the adaptation algorithm corresponds to a runtime model that can be adjusted dynamically to change the controller’s parameters (cf. Figure 8.5). In this case the runtime model is adjusted using feedforward control, that is by a mechanism external to the system (e.g., a human manager). In MIAC, the reference model is also a runtime model but adjusted using system identification methods. That is, the model is automatically adapted based on stimuli generated within the boundaries of the system (cf. Figure 8.6).

In the context of runtime software evolution, runtime models are important among others to represent evolution conditions, requirements and properties that must be assured, monitoring requirements and strategies, and to evolve the system or the evolution mechanism via model manipulation.

8.7.3 Runtime Monitoring

The *why* dimension of runtime software evolution characterizes reasons for evolving the system (cf. Figure 8.3). Runtime monitoring concerns the sensing and analysis of context information from the execution environment, which includes the system itself, to identify the need for evolution.

Context can be defined as any information useful to characterize the state of individual entities and the relationships among them. An entity is any subject that can affect the behavior of the system and/or its interaction with the user. Context information must be modeled in such a way that it can be pre-processed after its acquisition from the environment, classified according to the corresponding domain, handled to be provisioned based on the systems requirements, and maintained to support its dynamic evolution [896]. Based on this definition of context information, and from the perspective of runtime software evolution, runtime monitoring must support context representation and management to characterize the system's state with respect to its environment and evolution goals. Regarding context representation, operational specifications of context information must represent semantic dependencies among properties and requirements to be satisfied, evolution and V&V strategies, and the context situations that affect the evolution of the system. In highly uncertain situations, an important requisite is the representation of context such that its specifications can adapt dynamically, according to changes in requirements and the environment. Regarding context management, monitoring solutions must support every phase of the context information life cycle, that is context acquisition, handling, provisioning, and disposal. Context acquisition concerns the sensing of environmental information, handling refers to the analysis of the sensed information to decide whether or not to evolve the system, context provisioning allows evolution planers and executors to obtain environmental data that affect the way of evolving the system, and context disposal concerns the discard of information that is no longer useful for the evolution process. Moreover, to preserve context-awareness along the evolution process, monitoring infrastructures must be instrumented with self-adaptive capabilities that support the deployment of new sensors and handlers. For example, in our e-commerce scenario changes in the requirements due to the need for serving a new customer may imply new context types to be monitored.

8.7.4 Runtime Validation and Verification

Software validation and verification (V&V) ensures that software products satisfy user requirements and meet their expected quality attributes throughout their life cycle. V&V is a fundamental phase of the software evolution process [936]. Therefore, when the evolution is performed at runtime, V&V tasks must also be performed at runtime [823].

Aspects of runtime V&V that require special attention in the context of runtime software evolution include: (i) the dynamic nature of context situations; (ii) what to validate and verify, and its dependency on context information; (iii) where to validate—whether in the evolving system or the evolution mechanism; and (iv) when to perform V&V with respect to the adaptation loop implemented by evolution controllers. Researchers from communities related to SAS systems have argued for the importance of instrumenting the adaptation process with explicit runtime V&V tasks [221, 823]. In particular by integrating *runtime validators and verifiers*—associated with the planning phase of evolution controllers, and *V&V monitors*—associated with the monitoring process. The responsibility of runtime validators & verifiers is to verify each of the outputs (i.e., evolution plans) produced by the planner with respect to the properties of interest. The execution of an adaptation plan on a given system execution state implies a change of the system state. Therefore, the verification requirements and properties should be performed before and/or after instrumenting the plan. The responsibility of V&V monitors is to monitor and enforce the V&V tasks performed by runtime validators & verifiers.

8.8 Realizing Runtime Evolution in SMARTERCONTEXT

Figure 8.11 depicts a partial view of the software architecture of SMARTERCONTEXT using Service Component Architecture (SCA) notation. Further details about this architecture are available in [894]. SCA defines a programming model for building software systems based on Service Oriented Architecture (SOA) design principles [665]. It provides a specification for both the composition and creation of service components. *Components* are the basic artifacts that implement the program code in SMARTERCONTEXT. *Services* are the interfaces that expose functions to be consumed by other components. *References* enable components to consume services. *Composites* provide a logical grouping for components. *Wires* interconnect components within a same composite. In a composite, interfaces provided or required by internal components can be *promoted* to be visible at the composite level. *Properties* are attributes modifiable externally, and are defined for components and composites. *Composites* are deployed within an *SCA domain* that generally corresponds to a processing node.

Component *GoalsManager* has a twofold function. First, it allows system administrators to modify system goals (e.g. SLAs) at runtime. These changes in SLAs imply modifications in the monitoring requirements thus triggering the runtime evolution of SMARTERCONTEXT. Second, it enables system administrators to define and modify context reasoning rules as part of the evolution of the monitoring infrastructure at runtime. Components *ContextManager* and *DynamicMonitoringInfrastructure* implement the context monitoring functionalities of SMARTERCONTEXT. *ContextManager* includes software artifacts that integrate context information into context repositories, maintain and dispose existing contextual data, provide introspection support, and update the inventory of components managed dynamically as part of the runtime evolution process. *DynamicMonitoringInfrastructure* corresponds to the adaptive part of the monitoring infrastructure, and implements the context gathering, processing and provisioning tasks. In the case study described in Section 8.2, these tasks are performed by the components depicted within the dark gray box in Figure 8.11, *ContextGatheringAndPreprocessing* and *ContextMonitoring*. These components are deployed as part of the runtime evolution process to monitor the new bandwidth quality factor that is added after renegotiating the performance SLA. Component *MFLController* includes the artifacts that implement the feedback loop in charge of controlling the runtime evolution of our context management infrastructure.

The two *AdaptationMiddleware* components are an abstraction of FraSCAti [763] and QoS-CARE [820, 821], which constitute the middleware that provides the structural adaptation capabilities that allow the evolution of SMARTERCONTEXT at runtime.

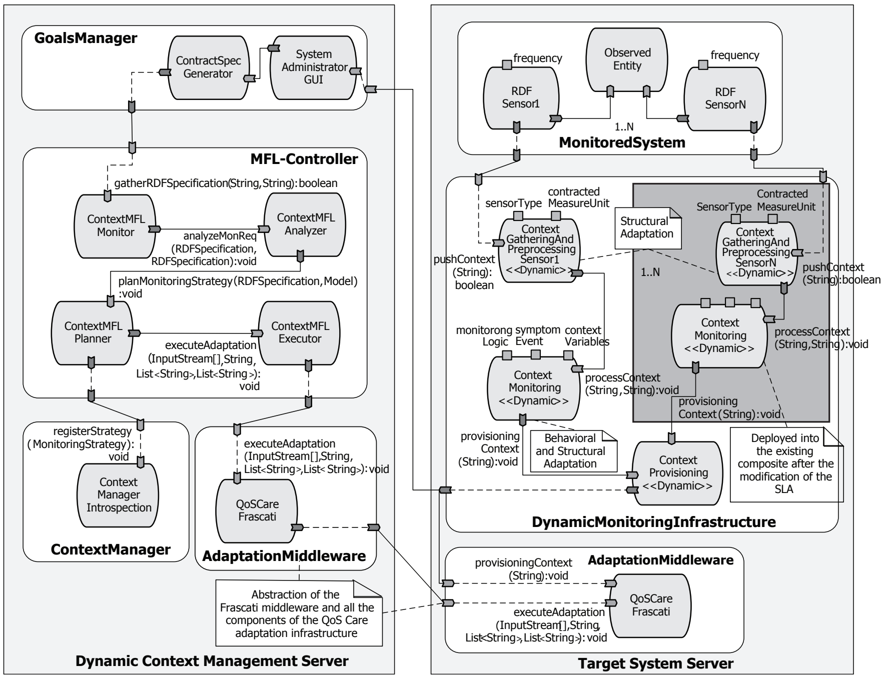

GoalsManager
ContractSpec Generator
System Administrator GUI
MFL-Controller
gatherRDFSpecification(String,String):boolean
ContextMFL Monitor
ContextMFL Analyzer
analyzeMonReq (RDFSpecification, RDFSpecification):void
planMonitoringStrategy(RDFSpecification,Model) :void
ContextMFL Planner
ContextMFL Executor
executeAdaptation (InputStream[],String, List<String>,List<String>): void
registerStrategy (MonitoringStrategy): void
Context Manager Introspection
ContextManager
executeAdaptation (InputStream[],String, List<String>,List< String>):void
QoSCare Frascati
AdaptationMiddleware
Abstraction of the Frascati middleware and all the components of the QoS Care adaptation infrastructure
Dynamic Context Management Server
frequency
Observed Entity
frequency
RDF Sensor1
RDF SensorN
1..N
MonitoredSystem
sensorType
contracted MeasureUnit
Context GatheringAnd Preprocessing Sensor1 <<Dynamic>>
Structural Adaptation
Contracted
SensorType MeasureUnit
Context GatheringAnd Preprocessing SensorN <<Dynamic>>
pushContext (String): boolean
1..N
pushContext (String):boolean
Context Monitoring <<Dynamic>>
processContext (String,String):void
monitorong Logic
symptom Event
context Variables
Context Monitoring <<Dynamic>>
processContext (String,String):void
provisioning Context(String):void
provisioning Context (String):void
Behavioral and Structural Adaptation
Context Provisioning <<Dynamic>>
Deployed into the existing composite after the modification of the SLA
DynamicMonitoringInfrastructure
provisioningContext (String):void
AdaptationMiddleware
QoSCare Frascati
executeAdaptation (InputStream[],String, List<String>,List<String>): void
Target System Server

Fig. 8.11: SMARTERCONTEXT's software architecture. The components depicted within the dark gray box represent software artifacts deployed dynamically as a result of the runtime evolution process.

8.8.1 Applying the MAPE-K Loop Reference Model

Composite *MFL-Controller* is an implementation of the MAPE-K loop reference model that enables our monitoring infrastructure with dynamic capabilities to evolve at runtime through the adaptation of (i) the set of context reasoning rules supported by the reasoning engine, (ii) the context monitoring logic that evaluates gathered context against monitoring conditions, and (iii) the context gathering and provisioning infrastructure.

The initial component of composite *MFL-Controller* is *ContextMFLMonitor*. This component receives the SLA specification in XML/RDF format from the user, creates an *RDFSpecification* object from the received specification, looks for a previous version of this SLA, stores the new SLA specification in its knowledge base, and finally provides component *ContextMFLAnalyzer* with two *RDFSpecification* objects that represent the new and former (if applicable) SLA specifications. SLA specifications in SMARTERCONTEXT are named *control objectives (COb) specifications* since they refer to the goals that drive the adaptive behavior of the system. [822]. SMARTERCONTEXT realizes COb specifications using Resource Description Framework (RDF) models [894]. RDF is a semantic web framework for realizing semantic interoperability in distributed applications [558].

The second component of the *MFL-Controller* composite is *ContextMFLAnalyzer*, which is in charge of analyzing changes in COb specifications, and specifying these changes in an RDF model. After analyzing changes in COb specifications, this component invokes *ContextMFLPlanner* and provides it with the new COb specification and the model that specifies the changes. If there is not previous COb specification, *ContextMFLAnalyzer* simply provides *ContextMFLPlanner* with the new COb specification and a null Model.

The third component of this MAPE-K loop is *ContextMFLPlanner*. This component is in charge of synthesizing new monitoring strategies as well as changes in existing ones. We define monitoring strategies as an object that contains (i) a set of implementation files for the SCA components to be deployed, the specification of the corresponding SCA composite, and two lists that specify SCA services and corresponding references. These services and references allow the connection of sensors exposed by the monitored third parties to context gatherers exposed by the SMARTERCONTEXT infrastructure, and context providers' gatherers exposed by third parties to context providers exposed by the SMARTERCONTEXT infrastructure; or (ii) a set of context reasoning rules to be added or deleted from the SMARTERCONTEXT's reasoning engine.

The last component of composite *MFL-Controller* is *ContextMFLExecutor*. This component invokes the services that will trigger the adaptation of the context monitoring infrastructure. The SMARTERCONTEXT monitoring infrastructure can be adapted at runtime by either (i) changing the set of context reasoning rules, (ii) modifying the monitoring logic, (iii) deploying new context sensors, and context gathering, monitoring and provisioning components. The case study presented in this chapter concerns mechanisms (ii) and (iii).

Table 8.2 summarizes the self-adaptive capabilities of SMARTERCONTEXT that allow its runtime evolution. The first column refers to changes in COb specifications that may trigger the evolution of SMARTERCONTEXT at runtime; the second column presents the type of control action used to adapt the monitoring infrastructure; the third column describes the evolution effect obtained on SMARTERCONTEXT after performing the adaptation process.

Table 8.2: Self-adaptive capabilities of SMARTERCONTEXT that support its evolution at runtime

3pt4pt

Changes in COb Specifications	Control Actions	Evolution Effects
Addition/deletion of reasoning rules	Parameters (i.e., RDF rules) affecting the behavior of SMARTERCONTEXT	Modified reasoning capabilities of the reasoning engine
Addition/deletion of context providers and/or consumers	Discrete operations affecting the SMARTERCONTEXT software architecture	Changes in the set of deployed context sensing, gathering, and provisioning components
Addition or renegotiation of system objectives	Parameters (i.e., arithmetic and logic expressions) affecting the behavior of SMARTERCONTEXT	Changes in existing monitoring logic
	Discrete operations affecting the SMARTERCONTEXT software architecture	Changes in the set of deployed context sensing, gathering, monitoring and provisioning components

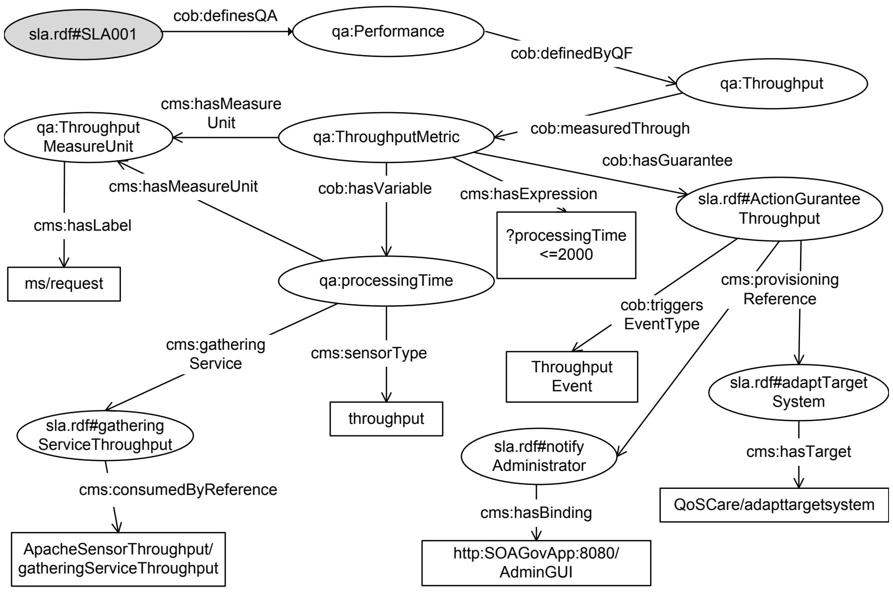
8.8.2 *Applying Requirements and Models at Runtime*

To control the relevance of the monitoring mechanisms implemented by SMARTERCONTEXT with respect to control objectives (e.g., the contracted QoS specified in SLAs), it is necessary to model and map these objectives explicitly to monitoring requirements. These models must be manipulable at runtime. This section illustrates the use of models at runtime to maintain operative specifications of requirements and evolving monitoring strategies during execution.

Control objectives (COb) specifications allow SMARTERCONTEXT to synthesize new and change existing monitoring strategies according to changes in contracted conditions. In the case study described in Section 8.2, COb specifications correspond to SLAs that not only define the contracted QoS and corresponding metrics, but also specify monitoring conditions associated with metrics, sensing interfaces and guarantee actions.

8.8.2.1 Control Objectives Specifications

Figure 8.12 represents, partially, a COb specification for the performance SLA that resulted from the first negotiation in our case study. This specification is a concrete instantiation of the *control objectives* ontology for QoS contracts in SMARTERCONTEXT. This ontology allows the specification of control objectives (e.g., SLAs) mapped to elements of both monitoring strategies and adaptation mechanisms represented by entities derived from the *context monitoring strategy (cms)* ontology [894].



Quality attributes vocabulary → qa: http://smartercontext.org/vocabularies/rdf/qa.rdf
Control objectives specification example → http://smartercontext.org/examples/thesis/sla-performance-SOACaseStudy-V1.rdf

Fig. 8.12: A control objectives specification example for the throughput quality attribute defined in the first negotiation of the performance SLA.

Namespace *qa:* corresponds to the vocabulary that characterizes quality attributes mapped to quality factors in the study. This version of the performance SLA defines a throughput quality factor, measured through a throughput metric (*qa:ThroughputMetric*) that is composed of a single variable (*qa:processingTime*). This variable is involved in the metric expression $?processingTime \leq 2000$, measured in terms of *ms/request* (as defined in the element *qa:ThroughputMeasureUnit*) and associated with a service identified as *sla.rdf#gatheringServiceThroughput*. The action guarantee *sla.rdf#ActionGuaranteeThroughput* associated with the throughput quality factor is associated with two provisioning references. The first one, *sla.rdf#adaptTargetSystem* is to invoke the service in charge of activating the adaptation process. The second one, *sla.rdf#notifyAdministrator*, is to inform business administrators about the violation of the contracted throughput conditions.

8.8.2.2 Synthesizing Monitoring Strategies at Runtime

In SMARTERCONTEXT monitoring strategies can be generated dynamically from COb specifications such as the one depicted in Figure 8.12. A monitoring strategy is defined as *DynamicMonitoringInfrastructure* composite (cf. the architecture depicted in Figure 8.11) that specifies components for context gathering, pre-processing, monitoring, and provisioning. These strategies are dynamic because SMARTERCONTEXT supports at runtime the modification of the monitoring logic, and, enabled by an architectural adaptation middleware, the deployment of new context management components.

Figure 8.13 illustrates, for the case study presented in this chapter, the generation of the *DynamicMonitoringInfrastructure* composite (cf. the highlighted composite in the same figure) from the COb specification presented in Figure 8.12. The RDF subgraphs associated with elements *sla.rdf#ActionGuaranteeThorughput* and *qa:ThroughputMetric* constitute the foundational elements for generating the *DynamicMonitoringInfrastructure* composite. The dotted connectors associate elements of the COb specification with the corresponding architectural artifact. For example, the connector labeled with number 1 indicates that component *ContextMonitoring* is dynamically generated from metric *qa:ThroughputMetric*.

When an existing SLA is renegotiated, the *GoalsManager* composite generates a new COb specification. Then, component *ContextMFLMonitor* defined in composite *MFL-Controller* gathers this specification, analyzes whether it corresponds to renegotiated SLA, and if so, generates a new plan with consists of the set of new context gathering and monitoring components to be deployed. Finally, the *ContextMFLExecutor* component executes the plan to deploy the new components.

8.9 Open Challenges

Many challenges remain open in the engineering of software systems with self-adaptive capabilities. Cheng et al. [177], as well as de Lemos et al. [221] characterize a comprehensive set of open questions and opportunities that are important to advance the field. The research roadmap by Cheng et al. focuses on development methods, techniques and tools required for the engineering of self-adaptive systems. This first roadmap groups challenges into four main topics: modeling dimensions, requirements, engineering, and assurances. The research roadmap by de Lemos et al. complements the first one while focusing on a different set of topics: design space, processes, decentralization of control loops, and practical runtime verification and validation (V&V).

Most difficult challenges in self-adaptation relate to the lack of effective methods for assuring the dynamic behavior of adaptive systems under high levels of uncertainty. In this realm control science and runtime models are research areas that deserve special attention. Control science can be defined as a systematic way to study certifiable V&V methods and tools to allow humans to trust decisions made

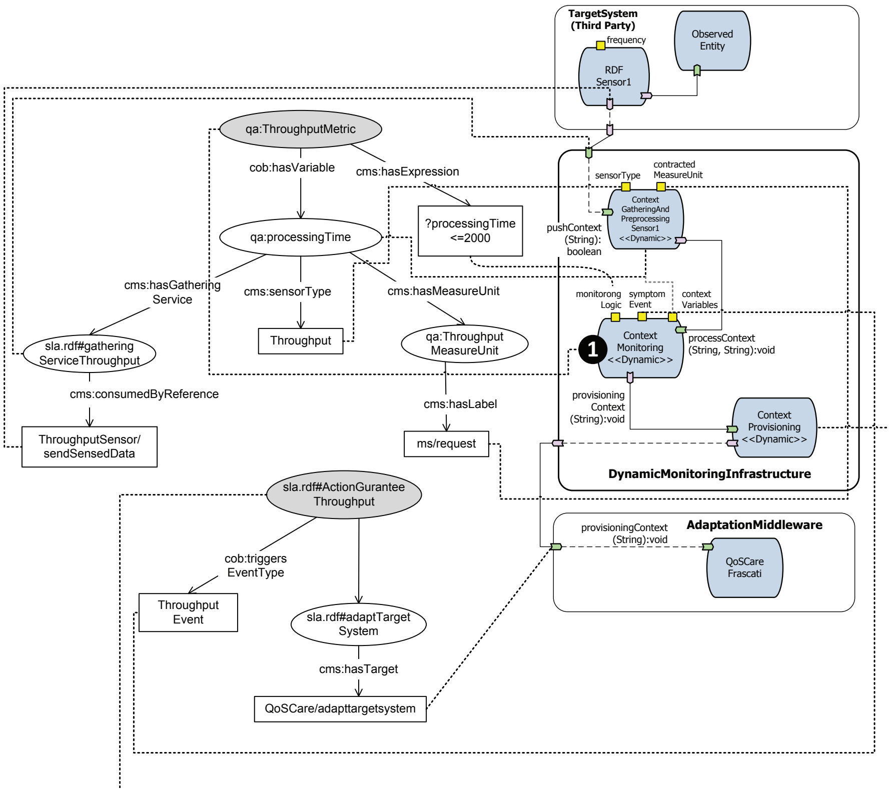

TargetSystem (Third Party)
frequency
RDF Sensor1
Observed Entity
qa:ThroughputMetric
cob:hasVariable
cms:hasExpression
qa:processingTime
?processingTime <=2000
sensorType
contracted MeasureUnit
Context GatheringAnd Preprocessing Sensor1 <<Dynamic>>
pushContext (String): boolean
cms:hasGathering Service
cms:sensorType
cms:hasMeasureUnit
monitorong Logic
symptom Event
context Variables
processContext (String, String):void
sla.rdf#gathering ServiceThroughput
Throughput
qa:Throughput MeasureUnit
Context Monitoring <<Dynamic>>
1
cms:consumedByReference
cms:hasLabel
provisioning Context (String):void
Context Provisioning <<Dynamic>>
ThroughputSensor/ sendSensedData
ms/request
DynamicMonitoringInfrastructure
sla.rdf#ActionGurantee Throughput
provisioningContext (String):void
AdaptationMiddleware
cob:triggers EventType
QoSCare Frascati
Throughput Event
sla.rdf#adaptTarget System
cms:hasTarget
QoSCare/adapttargetsystem

Fig. 8.13: Synthesizing monitoring strategies dynamically. The dashed connectors associate the element from the COb specification to the corresponding architectural artifact in the architecture.

by self-adaptive systems. In a 2010 report, Dahm identified control science as a top priority for the US Air Force (USAF) science and technology research agenda for the next 20 years [212]. Certifiable V&V methods and tools are critical for the success of self-adaptive systems. One systematic approach to control science for adaptive systems is to study V&V methods for the mechanisms that sense the dynamic environmental conditions and the target system behavior, and act in response to these conditions by answering the questions *what*, *when* and *how* to adapt [823].

Research on models at runtime study the exploitation of models available while the system executes. The goal is to provide effective mechanisms for complexity management in software systems whose behavior evolves at runtime [52]. Runtime models have been recognized as important enablers for the assurance of self-adaptive systems. We identified three subsystems that are key in the design of effective context-driven self-adaptation: the control objectives manager, the adaptation controller, and the context monitoring system [900]. These subsystems represent three levels of dynamics in self-adaptation that can be controlled through three feedback loops, i.e., the control objectives, the adaptation, and the monitoring feedback loops, respectively. We argue that runtime models provide abstractions that are crucial to support the feedback loops that control these three levels of dynamics. From this perspective, models at runtime could be developed specifically for each level of dynamics to support the control objectives manager, adaptation controller, and the monitoring system. At the control objectives level, models at runtime represent requirements specifications subject to assurance in the form of functional and non-functional requirements. At the adaptation level, models at runtime represent states of the managed system, adaptation plans and their relationships with the assurance specifications. At the monitoring level, models at runtime represent context entities, monitoring requirements, as well as monitoring strategies and their relationships with assurance criteria and adaptation models.

8.10 Conclusions

This chapter presented fundamental concepts of control and self-adaptive systems engineering, and their application to runtime software evolution. Departing from seminal aspects of "traditional" software evolution such as the change mini-cycle, Lehman's laws, and dimensions of software evolution, we discussed the complex dynamics of software systems and the way runtime software evolution can help deal with this complexity. Self-adaptation can be considered as short-term software evolution. Therefore, foundational concepts of self-adaptive software such as feedback, feedforward and adaptive control, the MAPE-K loop, ACRA reference architecture and self-* properties; and enabling mechanisms, such as requirements and models at runtime, context monitoring, and runtime V&V must be well understood by researchers, engineers and students interested in the evolution of highly dynamic and continuously running software systems.

It is important to point out that despite its benefits, runtime evolution is not always the best solution. We analyzed the need for runtime software evolution using the notion of its benefit cost ratio based on three selected variables: frequency of changes in requirements and environments, uncertainty, and off-line evolution cost. As part of this analysis we discussed trade-offs between runtime software evolution and the complexity added by the software artifacts required to automate software evolution tasks.

Rather than providing an exhaustive explanation of the application of self-adaptation and control theory to the engineering of runtime evolution mechanisms, or present new approaches to solve existing software evolution challenges, the goal of this chapter is to provide practitioners, researchers, and students with an overview of how the research work that is being conducted in the field of self-adaptive software relates to software evolution.