A white box perspective on behavioural adaptation

Bruni, Roberto; Corradini, Andrea; Gadducci, Fabio; Lluch Lafuente, Alberto; Vandin, Andrea

Published in:
Software, Services, and Systems

Link to article, DOI:
10.1007/978-3-319-15545-6_32

Publication date:
2015

Document Version
Peer reviewed version

A White Box Perspective on Behavioural Adaptation*

Roberto Bruni[1], Andrea Corradini[1], Fabio Gadducci[1], Alberto Lluch Lafuente[2], and Andrea Vandin[3]

[1] Dipartimento di Informatica, Università di Pisa, Italy
[2] DTU Compute, Technical University of Denmark, Denmark
[3] Electronics and Computer Science, University of Southampton, UK

Abstract. We present a white-box conceptual framework for adaptation developed in the context of the EU Project ASCENS coordinated by Martin Wirsing. We called it CoDa, for Control Data Adaptation, since it is based on the notion of *control data*. CoDa promotes a neat separation between application and adaptation logic through a clear identification of the set of data that is relevant for the latter. The framework provides an original perspective from which we survey a representative set of approaches to adaptation, ranging from programming languages and paradigms to computational models and architectural solutions.

Keywords: Adaptation, Self-*, Autonomic Computing, Programming Languages, Software Architectures, Computational Models, Computational Reflection

1 Introduction

Self-adaptive systems have been widely studied in several disciplines like Biology, Engineering, Economy and Sociology. They have become a hot topic in Computer Science in the last decade as a convenient solution to the problem of mastering the complexity of modern software systems, networks and architectures. In particular, self-adaptation is considered a fundamental feature of *autonomic systems*, often realized by specialized self-* mechanisms like self-configuration, self-optimization, self-protection and self-healing, as discussed for example in [41].

The literature includes valuable works aimed at capturing the essentials of adaptation both in the most general sense (see, e.g., [49]) and more specifically fields such as software systems (see, e.g., [68,13,52,5,65]) providing in some cases very rich surveys and taxonomies. A prominent and interesting example is the taxonomy of concepts related to self-adaptation presented in [68], whose authors remark the highly interdisciplinary nature of the studies of such systems. Indeed, just restricting to the realm of Computer Science, active research on self-adaptive systems is carried out in Software Engineering, Artificial Intelligence, Control Theory, and Network and Distributed Computing, among others.

* Research supported by the European projects IP 257414 ASCENS and STReP 600708 QUANTICOL, and the Italian project PRIN 2010LHT4KM CINA.

Despite all these classification efforts, there is no agreement on the *conceptual* notion of adaptation, neither in general nor for software systems. Lofti Zadeh noticed in [79] that *"it is very difficult—perhaps impossible—to find a way of characterizing in concrete terms the large variety of ways in which adaptive behavior can be realized"*. Zadeh's concerns were conceived in the field of Control Theory but as many authors agree (e.g., [65,68,5,49]), they are valid in Computer Science as well. One reason for Zadeh's lack of hope in a concrete unifying definition of adaptation is the attempt to subsume two aspects under the same hat: the *external* manifestations of adaptive systems, and the *internal* mechanisms by which adaptation is achieved. We shall refer to the first aspect as the *black-box* view on adaptation, and to the second aspect as the *white-box* one.[4]

Actually, in the realm of Software Engineering there are widely spread informal definitions, according to which a software system is called "self-adaptive" if it *"modifies its own behavior in response to changes in its operating environment"* [60], where such "environment" has to be understood in the widest possible way, including both the external environment and the internal state of the system itself. Typically, such changes are applied when the software system realizes that *"it is not accomplishing what the software is intended to do, or better functionality or performance is possible"* [47]. Such definitions can be exploited to measure what is often called the *degree of adaptivity*, i.e., to estimate the system robustness under some conditions. This approach can be traced back to Zadeh's proposal [79], but has been later adopted by many other authors (e.g., [58,39]).

The problem is that almost any software system can be considered self-adaptive according to the above definitions, as it can *modify its behaviour* (e.g., by redirecting the control flow) as a *reaction to a change in its context of execution* (like the change of variables). Thus, such definitions, concerned with the *observational* perspective only, are of difficult applicability for distinguishing adaptive systems from "non-adaptive" ones. Also, they are of little use for design purposes, where separation of concerns, modularization, reuse are crucial aspects.

The development and success of many emergent Computer Science paradigms is often strongly supported by the identification of key principles around which the theoretical aspects can be conveniently investigated and fully worked out. For example, in the case of distributed computing, there have been several efforts in studying the key primitives for communication, including mechanisms for passing communication means (name mobility) or entire processes (code mobility), which led to a widely understood theory of mobile process calculi. There is unfortunately no such agreement concerning (self-)adaptation, as it is not clear what are the characterizing structural features that distinguish such systems from plain ones.

Summarizing: (i) existing definitions of adaptation (and also *adaptivity* and *adaptability*) are not always useful in pinpointing adaptive systems, but they allow to discard many systems that certainly are not, and (ii) their focus is often more on the issue of *how much* a system adapts than *in which manner*.

[4] The black- and white-box *perspective* should not be confused with the white- and black-box component adaptation *techniques* as discussed, e.g., in [12], where *black* refers to exploiting the interface of a component and *white* to exploiting its internals.

Contributions and structure. The paper presents a conceptual framework for adaptation by means of a simple structural criterion. This framework, introduced in Section 2, is called CoDa, COntrol Data Adaptation. Our contribution is a definition of adaptation that is applicable to most approaches in the literature, and in fact it is often coincident with them once it is instantiated to each approach. Also, we aim at a *separation of concerns* to distinguish changes of behaviour that are part of the application logic from those where they realize the adaptation logic, calling "adaptive" only those systems capable of the latter. More precisely, we propose concrete answers to basic questions like *"is a software system adaptive?"* or *"where is the adaptation logic in an adaptive system?"*. We take a *white-box* perspective that allows us to inspect, to some extent, the internal structure of a system. Moreover, we provide the designer with a criterion to specify where adaptation is located and, as a consequence, what parts of a system have to be adapted, by whom and how. Note that while adaptation can be concerned with a single component as well as with a whole system, we will not push this distinction and will address both situations: the case will be evident by the context.

The second part of the paper (Sections 3–5) is devoted to a *proof of concept*: we overview several approaches to adaptation and validate how the CoDa definition of adaptation is applied to them. This part of the paper is organized according to different pillars of Computer Science: *architectural* approaches (Section 3), *foundational* models (Section 4), and *programming* paradigms (Section 5). Approaches that cover more than one of such aspects are discussed only once.

It is worth remarking that it is not the programming paradigm, the architecture or the underlying foundational model what makes a system adaptive or not. For example, adaptive systems can be programmed in any language, exactly like object-oriented systems can in imperative languages, albeit with some effort. However, it is beyond the scope of this paper to discuss approaches that do not address adaptation in an explicit way, even if they might do so implicitly.

Section 6 overviews other surveys and taxonomies that address the same aim of our work. Finally, Section 7 concludes the paper and discusses future research.

Our work would not be the same without the support and insights from Martin Wirsing. It was indeed conceived in early meetings of the ASCENS project, coordinated by Martin. The main questions under discussion were the meaning of adaptation and its formalization. We presented some preliminary ideas essentially based on the use of logical reflection in algebraic specifications. Though sharing our passion for such disciplines and understanding our points Martin suggested warned us about the difficulties of meta-programming techniques and encouraged us to consider other approaches, including those proposed by other teams of the project. This lead us to investigate the essence of adaptation, and resulted first in the shorter, less inclusive version of this paper appeared as [20], and ultimately in the present work. We would like to express infinite gratitude to Martin, for his tenacious guidance, his calm patience and his pointed intuitions during all these beautiful years of fruitful research collaborations.

2 When is a software component adaptive?

The behavior of a software component is governed by a program, and, according to the traditional view (e.g., [78]), a program is made of *control* (i.e., algorithms) and *data*. This basic view of programs is sufficient for the sake of introducing our approach. CoDa requires to make explicit that the behaviour of a component depends on some *control data* that can be changed to *adapt* it. At this level of abstraction we are not concerned with the structure of control data, the way they influence the behaviour of the component, or the causes of their modification.

Our definition of adaptation is then very straight: *Given a component with a distinguished collection of control data,* adaptation *is the runtime modification of such control data.* From this definition we can easily derive several others. A component is *adaptable* if its control data may be modified at runtime, it is *adaptive* if its control data are actually modified at runtime in some execution, and it is *self-adaptive* if it modifies its own control data at runtime.

The CoDa point of view is in line with other white-box perspectives on adaptation as we discuss in Section 6. Our goal is to show that the conceptual view of CoDa enjoys two key properties: concreteness and generality.

Concreteness. Any definition of adaptation should face the problem that the judgement whether a system is adaptive or not is often subjective. From the CoDa perspective, this is captured by the fact that the collection of control data of a component can be defined, at least in principle, in an arbitrary way, ranging from the empty set ("the system is not adaptable") to the collection of all the data of the program ("any data modification is an adaptation"). As a concrete example, consider the following conditional statement:

if `the_hill_is_too_steep` **then** `assemble_with_others` **else** `proceed_alone`

Can it be interpreted as a form of adaptation? From a black-box perspective the answer is "it depends". Indeed, the above statement is typical of controllers for robots operating collectively as swarms and having to face environments with obstacles (see, e.g., [59]). As some authors observe [37] *"obstacle avoidance may count as adaptive behaviour if [...] obstacles appear rarely. [...] If the "normal" environment is [...] obstacle-rich, then avoidance becomes [...] normal behaviour rather than an adaptation"*. In sum, the above conditional statement can be a form of adaptation in some contexts but not in others.

Now, suppose that the statement is part of the software controlling a robot, and that `the_hill_is_too_steep` is a boolean variable set according to the value returned by a sensor. Then, in our framework the change of behaviour caused by a modification of its value is considered as an adaptation or not depending on if `the_hill_is_too_steep` is considered as part of the control data or not.

Such a boolean variable is not in itself a datum obtained by a sensor: it is controlled by an adaptation logic that changes its value when a given threshold is reached in the information received by the sensors: thus, our control data do not by necessity coincide with sensor data. In more general terms, the difference is going to be made explicit e.g. when we will instantiate our CoDa approach

in the context of computational models that support meta-programming or reflective features, where a program-as-data paradigm holds: the issue is tackled in Section 4.2 and in the summary of forms assumed by control data in Fig. 1.

Summing up, the above question (i.e., *"can it be interpreted as a form of adaptation?"*) can be answered only after a clear identification of the control data. This means that from the white-box perspective of CoDa the answer is still "it depends" as it is for the black-box case. However, there is a fundamental difference: the responsibility of declaring which behaviours are part of the adaptation logic is passed from the observer of the component to its designer. Ideally, a sensible collection of control data should be chosen to enforce a separation of concerns, allowing to distinguish neatly, if possible, the activities relevant to adaptation (those that affect the control data) from those relevant to the application logic only (that should not modify the control data).

Generality. Any definition of adaptation should be general enough to capture the essence of the most relevant approaches to adaptation proposed in the literature. The generality of CoDa is witnessed by the discussion of Sections 3–5 where we overview several approaches to adaptation, pointing out for each of them what we consider the natural candidates for control data. More explicitly, the criterion that we shall use for determining such data is the following: *a system designed according to one of such approaches manifests an adaptation exactly when the corresponding control data are modified.*

Adaptive systems are realized by resorting to a variety of computational models and programming paradigms. The nature of control data can thus vary considerably: from simple configuration parameters to a complete representation of the program in execution that can be modified at runtime.

The variety of formalisms makes it hard to compare approaches with each other, unless one manages to map them into a unifying model of computation (which is far beyond the scope of this paper). However, for the sake of a brief discussion we enrich our intuitive view of a system as made of control, control data and ordinary data, with additional features such as the system's *architecture* (in a general sense, including the interconnection of components, communication stacks, workflows, etc.), and the *adaptation strategy* used to enact adaptation. Moreover we shall assume that the behavior of the system or component (i.e., its control) may be structured into sub-parts that we call *operation modes*.

Such simple perspective on adaptive systems helps us in classifying the main approaches surveyed in this paper as depicted in Figure 1. Symbol "*" is used to denote generic approaches that propose reference models where control data depends on concrete instances of the approach. The table also contains the control data as-it-is and the section where the approach is discussed.

Such classification has several advantages: (i) It provides a criterion that is orthogonal to those of the surveys and taxonomies discussed in Section 6 and to the classification by research areas along which we structure Sections 3–5. (ii) It allows us to relate approaches presented independently and in different areas but sharing, essentially, the same category of control data. This is, e.g., the case of the approaches based on modes of operation proposed by the Software Engineering

	CONTROL DATA (as-it-is)	CONTROL DATA (class)	Section
[40]	*	*	3.1
[24]	*	*	3.1
[77]	*	*	3.1
[43]	*	*	3.2
[62]	*	*	4.3
[11]	adaptation coordination strategies	adaptation strategy	4.1
[48]	adaptation rules	adaptation strategy	5.3
[15]	architecture	architecture	3.1
[46]	architecture	architecture	3.2
[60]	architecture	architecture	3.2
[66]	module stack	architecture	3.2
[22]	current workflow	architecture	3.2
[7]	connectors	architecture	3.2
[11]	architecture	architecture	4.1
[76]	effector channel	architecture	4.3
[48]	set of activities	architecture	5.3
[61]	entire programs	entire program	4.1
[55]	rewrite rules	entire program	4.2
[35]	processes	entire program	4.3
[30]	processes	entire program	4.3
[28]	features	operation mode	4.1
[53]	regions	operation mode	4.1
[83]	operation mode	operation mode	4.1
[1]	active configuration	operation mode	4.1
[72]	active configuration	operation mode	4.1
[19]	control proposition	operation mode	4.1
[82]	steady state programs	operation mode	4.1
[42]	state space zones	operation mode	4.1
[33]	graph rewrite rules	operation mode	4.2
[80]	base level Petri net	operation mode	4.3
[51]	adaptor processes	operation mode	4.3
[16]	adaptable (local) processes	operation mode	4.3
[69]	context stack	operation mode	5.1
[36]	advices	operation mode	5.2
[44]	policies	operation mode	5.3

Fig. 1. Summary of some of the control data forms discussed.

community with paradigm-oriented approaches and by the Theoretical Computer Science community with automata and process-algebraic approaches. (iii) It allows us to compare approaches apparently similar (and falling in the same section) but based on different categories of control data. For instance, in some process-algebraic approaches the control data may reside in the communication topology or in the entire program. Note that the classification depends on the envisioned conceptual computational formalisms where we map the approaches. We have proposed a simple one to illustrate a possible way of exploiting the notion of control data for comparison purposes.

3 Architectural approaches to adaptation

Several contributions to the literature describe architectural approaches to autonomic computing and self-adaptive software systems. In this section we survey some of such proposals, organizing the discussion around two main themes: reference models (Section 3.1) and reconfiguration-based approaches (Section 3.2).

3.1 Reference Models for Adaptation

In this section we review here, among others, two influential reference models for adaptive and self-adaptive systems: MAPE-K [40] and FORMS [77]. Both approaches propose general guidelines for the architecture of (self-)adaptive systems, the first one based on the presence of a control loop, the second one on the use of computational reflection. The identification of control data at this level of abstraction can only be very generic, as concrete instances may realize the reference models in significantly different ways.

The first reference model we consider is MAPE-K (Monitor, Analyse, Plan, Execute, Knowledge), introduced in the seminal [40]. A self-adaptive system is made of a component implementing the application logic, equipped with a control loop that monitors the execution through suitable sensors, analyses the collected data, plans an adaptation strategy, and finally executes the adaptation of the managed component through some effectors; all the phases of the control loop access a shared knowledge repository. The managed component is considered to be an adaptable component, and the system made of the component and the manager implementing the control loop is considered a self-adaptive component.

The conceptual role of the control loop induces a natural choice for the control data: while in the monitor phase a wide range of data from the managed component may be sensed, the control data are those that are modified by the execute phase of the control loop. Thus the control data of a managed component is (explicitly or implicitly) available via the interface it offers to its manager, which can use it to enact its control loop, as shown in Fig. 2. Clearly, the concrete structure of control data (e.g., variables, policies, ...) depends on the specific instance of the MAPE-K model and on the computational model or programming language used, as discussed in the next two sections. The construction can be iterated, as the manager itself can be an adaptable component.

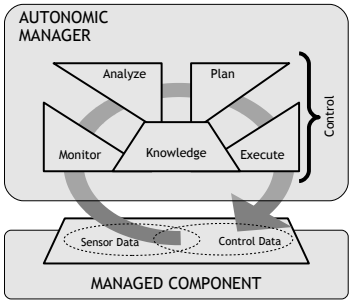

Fig. 2. Control data in MAPE-K.

Concrete instances of this scenario can be found, among others, in [11,48,23]. For example, in the latter, components follow plans to perform their tasks and re-planning is used to overcome unpredicted situations that may make current plans inefficient or impossible to realize. A component in this scenario

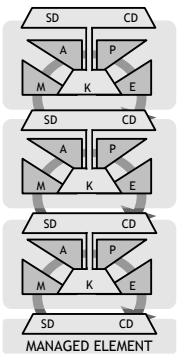

Fig. 3. Tower of adaptation.

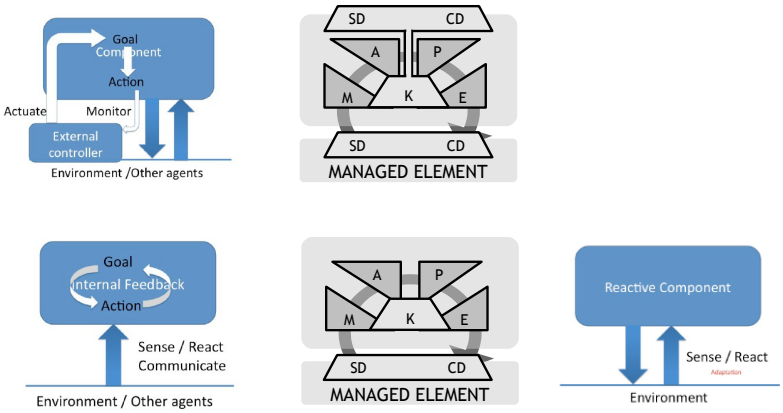

Fig. 4. External (top-left) and internal (bottom-left) control loop patterns and their presentation in terms of the MAPE-K model (center), and the reactive pattern (right).

can be adaptable, having a manager which devises new plans according to changes in the context or in the component's goals. In turn, this planning component might itself be adaptable, with another component that controls and adapts its planning strategy, e.g., on the basis of a tradeoff between optimality of the plans and computational cost of the planning algorithms. In this case, the planning component (that realizes the control loop of the base component) exposes some control data (conceptually part of its knowledge), thus enabling a hierarchical composition that allows building towers of adaptive components (Fig. 3).

The MAPE-K control loop is very influential in the autonomic computing community, but control loops in general have been proposed and extensively studied also by others as a key mechanism for achieving self-adaptation in software systems, also on the basis of the crucial role they play in engineering disciplines like Control Theory. An interesting survey of several types of control loops is presented in [18], which among others identifies the *Model Reference Adaptive Control loop*, where the control loop is fed with a model of the controlled component, and the *Model Identification Adaptive Control loop*, where the control loop tries to infer such a model directly from the behaviour of the component.

Typical control loop patterns are also proposed in [24], which presents a taxonomy of design patterns for adaptation (see Fig. 4). In the *internal control loop* pattern, the manager is a wrapper for the managed component and it is not adaptable. Instead, in the *external control loop* pattern, the manager is an adaptable component that is connected with the managed component. The distinction between external and internal control loops is also discussed in [68], where it is stressed that internal control loops offer poor scalability and maintainability due to the intertwining of the application and the adaptation

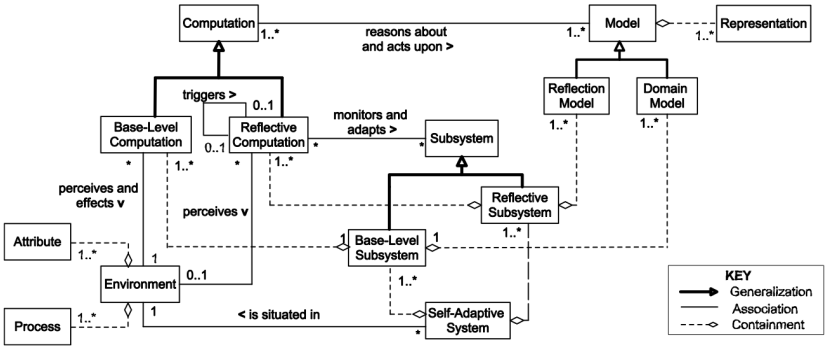

Fig. 5. The FORMS reference model.

logic. Indeed this contradicts the separation-of-concerns principle that the authors (and many others) promote as key feature of self-adaptive systems. Like for MAPE-K, also for these control-loop centered approaches to adaptivity a precise identification of control data is only possible in concrete instances.

The taxonomy of [24] includes a third pattern called *reactive pattern* that describes *reactive* components capable of modifying their behavior in reaction to an external event, without any control loop (or, equivalently, with a degenerate, "empty" control loop). In order to apply our definition of adaptation as *runtime modification of control data* to a reactive system of this kind, one could simply identify as control data those data that, when modified by sensing the environment, cause an adaptation of the system. This is a good example of the generality of our definition of adaptation, which is applicable also to such quite extreme case.

The reference model in [6] promotes *computational reflection* as a necessary criterion for any self-adaptive software system. Reflection implies the presence, besides of base-level components and computations, of meta-level subsystems and meta-computations that act on a meta-model. Meta-computations inspect and modify the meta-model that is causally connected to the base-level system, so that changes in one are reflected in the other. The authors argue that most methodologies and frameworks for the design and development of self-adaptive systems rely on some form of reflection, even if not explicitly. Building on these considerations, they introduce the FOrmal Reference Model for Self-adaptation (FORMS) [77], providing basic modeling primitives, and relationships among them, for the design of self-adaptive systems (cf. Fig. 5), and making explicit the presence of reflective (meta-level) subsystems, computations and models.

The goals of [6] are not dissimilar from ours, as they try to capture the essence of self-adaptive systems, identifying it in computational reflection (one of the key features of self-adaptive systems according to [52] as well). The FORMS modeling primitives can be instantiated and composed in a variety of ways. For example, [77] provides one example that conforms to the MAPE-K reference model and another one that follows an application-specific design.

A precise identification of control data depends on the specific instance of the approach, and more precisely on the way modifications to the meta-level affect the base level, causing an adaptation. In instances featuring some kind of hot-linking from the meta- to the base-level component, the meta-level itself can be considered as control data. Otherwise, in general, control data will be identified at the boundary between the meta-level and the base-level components.

3.2 Reconfiguration-based Approaches to Adaptation

Several approaches to the design of (self-)adaptive systems look at a system as a network of components, suitably arranged in a logical or physical topology that constraints the interactions or communications among components. Adaptations in this context are typically realized via *reconfigurations*, which can range from the replacement of a single component to local or even global changes to the interaction topology. Usually such reconfigurations do not modify the functionalities of the individual components, but only the way they are connected and/or interact with each other (see the survey [15], summarized in Section 6, and [46]). Therefore the control data in these approaches can be identified with the interconnection topology itself, which depending on the approaches can be made of channels, connectors, gates, protocol stacks, links, and so on.

A first example is the approach presented in [60], where dynamic software architecture has a dominant role. The proposed methodology combines an Adaptation Management loop, which is essentially a distributed, agent-based MAPE-K control loop, with an Evolution Management loop. In the latter, an architectural model is maintained at runtime, that describes the running implementation and that plays the role of our control data. In fact the architectural model, made of components and connectors, can be modified by the control loop, by adding or removing components or connectors or by changing the topology. An Architecture Evolution Manger mediates the changes of the architectural model and maintains the consistency between the model and the running implementation.

The Ensemble system [66] is a network protocol architecture conceived with the aim of facilitating the development of adaptive distributed applications. The main idea is that each component of the application relies on a reconfigurable stack made of simple micro-protocol modules, which implement different component-to-component communication features. The module stack imposes a layered structure to the communication infrastructure which is used to guide its adaptation. For instance, adaptation can be triggered in a bottom-up way, when a layer n discovers some environmental changes that require an adaptation. Then the module at layer n may be adapted and, if not possible, the adaptation request is propagated to the upper layer $n+1$. Such structure is also exploited when a coordinated, distributed adaptation is needed, which is tackled by the *Protocol Switching Protocol*, one the key features of the approach. The protocol is initiated by a global coordinator that sends the notification of the need of adaptation to each component. Within each component the notification is propagated through the protocol stack, so that each layer applies the necessary actions. Adaptation can happen at different

points. In particular it may affect the components participating to the distributed application (or to groups within it) or the communication infrastructure (i.e., the module stack). Hence, generally speaking, the set of components, their state and the module stack form the control data of the adaptive application.

The authors of [43] discuss how to apply this model-based approach to Model-Integrated Computing to adaptive systems. Adaptation is mainly reconfiguration followed by automatic deployment, triggered at runtime by the user or by the system as a reaction to some events. In the proposed case study, a simple finite-state automaton determines the transitions from one behaviour to another: here, the natural choice of control data consists of the states of the automaton.

A life-cycle for service-based applications where adaptation is a first-class concern is defined in [22]. Such life-cycle continues during runtime to cope with dynamic requirements and the corresponding adaptations. In addition to the life-cycle, [22] focuses on the identification of a number of design principles and guidelines that are suitable for adaptable applications. Essentially, adaptation is understood as the modification of the workflow implementing a service-based application, from substituting individual services by equivalent ones, to recomposing a piece of the workflow to obtain an equivalent result. Therefore, roughly speaking, the current workflow is the control data of the service-based applications.

In the architectural approach of [7] a system specification has a two-layered architecture to enforce a separation between computation and coordination. The first layer includes the basic computational components and their interfaces, while the second one is made of connectors (called *coordination contracts*) that link the components to ensure the required system's functionalities. Adaptation in this context is obtained by reconfiguration, which consists of removal, addition or replacement of both base components and connectors among them. The possible reconfigurations of a system are described declaratively with suitable rules, grouped in *coordination contexts*: such rules can be either invoked explicitly, or triggered automatically when certain conditions are satisfied. In this approach, as adaptation is reconfiguration, the control data consist of the whole two-layered architecture, excluding the internal state of the computational components.

4 Computational Models for Adaptation

Computational reflection is widely accepted as one of the key instruments to build self-adaptive systems (cf. [52,31]). Indeed computational paradigms equipped with reflective, meta-level or higher-order features, allow one to represent programs as first-class citizens. In these cases adaptation emerges, according to our definitions, if the program in execution is represented in the control data of the system, and it is modified during execution. Prominent examples of such formalisms are, e.g., rewrite theories with logical reflection like rewriting logic [54] or process calculi with higher-order or meta-level aspects like HO π-calculus [71]. Systems represented within these paradigms can realize self-adaptation in a straightforward manner. Of course, computational reflection assumes different forms and, despite

of being a very convenient mechanism, it is not strictly necessary: as we argued in Section 1 any programming language can be used to build a self-adaptive system.

We outline in this section some rules of thumb for the choice of control data within some well-known computational formalisms (deferring programming paradigms and languages to Section 5). In doing so, we restrict the attention to computational models that have been purposely introduced to represent adaptation and we point out how they can be used for modeling the behavior of self-adaptive systems. In addition, we survey a representative set of models that have been conceived with the specific purpose of modeling self-adaptive systems and supporting their formal analysis. We structure the presentation along three main strands: automata-like computational models (Section 4.1), declarative, rule-based computational models (Section 4.2), and computational models from the concurrency theory field (Section 4.3).

4.1 Automata-based Approaches to Adaptation

In many frameworks for the design of adaptive systems the base-level system has a fixed collection of possible behaviours (or behavioural models), and adaptation consists of passing from one behaviour to another. Some of the approaches discussed in this section achieve this by relying on a multi-layered structure reminiscent of hierarchical state machines and automata.

A first example of this tradition are the Adaptive Featured Transition Systems (A-FTS) of [28], which were introduced for the purpose of model checking adaptive software (with a focus on software product lines). A-FTSs are a sort of transition systems where states are composed by the local state of the system, its configuration (set of *active features*) and the configuration of the environment. Transitions are decorated with executability conditions that regard the valid configurations. Adaptation corresponds to reconfigurations (changing the system's features). Hence, in terms of our white-box approach, reconfigurable system features play the role of control data. The authors introduce the notion of *resilience* as the ability of the system to satisfy properties despite of environmental changes (which essentially coincides with the notion of black-box adaptivity of [39]). Properties are expressed in AdaCTL, a variant of the computation-tree temporal logic CTL.

Another example of layered computational structures are S[B] systems [53], a model for adaptive systems based on 2-layered transitions systems. The base transition system B defines the ordinary behavior of the system, while S is the adaptation manager, which imposes some regions (subsets of states) and transitions between them (adaptations). Further constraints are imposed by S via adaptation invariants. Adaptations are triggered to change *region* (in case of local deadlock). Such regions, hence, form the control data of the system according to our white-box approach. The paper also formalizes notions of *weak* and *strong* adaptability, defined as the ability to conclude a triggered adaptation in some or all possible behaviors, respectively, and characterized by suitable CTL formulae.

Mode automata [50] have been also advocated as a suitable model for adaptive systems. For example, the approach of [83] represents adaptive systems with

two layers: a *functional layer*, which implements the application logic and is represented by state machines called *adaptable automata*, and an *adaptation layer* that implements the adaptation logic and is represented with a mode automaton. Adaptation here is the change of mode, and these are the control data of this approach. The approach considers three kinds of specification properties: *local* (to be satisfied by the functional behavior of one particular mode, not involving adaptation), *adaptation* (to be satisfied by adaptation phases, i.e., transitions between modes), and *global* (to be satisfied by all behaviors). An extension of linear-time temporal logic (LTL) called *m*LTL is used to express such properties.

Overlap adaptations [11] arise in long-running open and dynamic distributed applications where components can be removed, added or replaced with a certain frequency. Under these premises, it is clear that the set of components of the application corresponds to its control data. An overlap adaptation occurs when the execution of *old* components (i.e., components that need to be adapted) overlaps with the execution of *new* components (i.e., adapted components). This overlap introduces non-trivial issues but is required in order to adapt the whole application in a distributed manner without stopping it.

The authors identify several kinds of overlap adaptations which vary in the kind of allowed interactions between old and new components. The main concern of the approach is verifying the correctness of adaptations. For this purpose the approach relies on the concept of *transitional adaptation lattices*, roughly, diamond-shaped graphs whose nodes are automata and whose transitions correspond to atomic adaptation actions (cf. Fig. 6). Each automaton represents the behavior of the whole system in some state. The *top* automaton corresponds to the system before adaptation starts, while the *bottom* automaton corresponds to the system when adaptation ends. The diamond shape of the lattice implicitly imposes a confluent behavior of individual atomic adaptations.

Actually, the approach considers a finer granularity of components in terms of *fractions*, which are essentially the local instances of components in process locations, introducing a combinatorial explosion in the size of the lattices which has a negative impact in the effort required in their analysis. To mitigate this the authors propose a framework based on particular architectures and coordination protocols, where some specialized modules drive the adaptation phase through designated paths in the adaptation lattices. This implicitly introduces a higher-level adaptation since a system may vary the strategy of such modules according to various factors: the control data of the system correspond to such strategies.

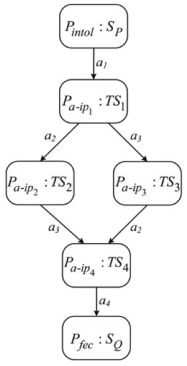

Fig. 6. Adaptation lattice.

Another example of labelled transition system variant used for modeling self-adaptive systems are the *Synchronous Adaptive Systems* of MARS [1,72], where systems are modeled as sets of modules, each having a set of configurations. At runtime only one configuration is active. Adaptation consists on changing the

active configuration, selected according to the configuration and environment status. Control data are thus those that determine the active configuration.

While the "programs-of-programs" spirit can raise scalability and complexity issues, the layered structure of some of the above models can be exploited to study adaptive systems compositionally. The authors of [82] propose a technique to verify properties of adaptive systems in a modular way. Adaptive programs are modeled with *n-plex adaptive programs* which are essentially sets of finite state machines, some of which representing *steady state programs* [4] and the rest representing adaptation transitions between those programs. The structure of an n-plex adaptive program makes explicit the separation of functional concerns (realized by steady state programs) and adaptation concerns (realized by adaptation transitions), which is exploited to reason about such systems in a modular way. Clearly, the separation of concerns coincides with the spirit of CoDa. In particular, control data here are the individual steady state programs.

This separation of concerns has its counterpart in the property specification language used, *Adapt-operator extended LTL* (A-LTL) [81]. A-LTL extends LTL with an operator that does not provide more expressive power but allows to express properties of adaptive systems more concisely. With respect to similar approaches, the modular verification phase exploits the separation of concerns and the assume/guarantee paradigm in order to avoid the state explosion problem, thus providing a more scalable solution. For instance, this allows the authors to tackle *transitional* properties of adaptation (e.g., graceful adaptation, hot-swapping adaptation, etc.) in an efficient manner.

Structuring the behavior of adaptive system is a major concern in [42]. The authors identify four main modes of operation (called *state space zones*) in an adaptive system: the *normal* behavior zone (the system operates as expected), the *undesired* behavior zone (the system has violated some constraint and needs to be adapted), the *invalid* behavior zone (the system has violated some constraint and cannot be adapted), and the *adaptation* behavior zone (the system is adapting to re-enter the normal behavior zone). The work is motivated by the necessity of shifting the focus to behavioral aspects of adaptation, as evidenced in previous experiences of the authors that were mainly concerned with architectural aspects [77]. In this approach, hence, the control data are those used to characterize the state space zones. The approach is validated with a case study of a decentralized adaptive traffic control system using timed automata and a timed extension of CTL. The authors distinguish two different adaptation capabilities (from the black-box perspective): *flexibility* (ability to adapt to changing environments, e.g., to improve performance) and *robustness* (ability to recover from failures).

Some of the above approaches rely on logical reasoning mechanisms to prove properties of adaptation. To this end, base steady programs are annotated with the properties they ensure (cf. the above discussed adaptation lattices [11]). This idea of specification-carrying programs is investigated in [61]. Suitable semantical domains aimed at capturing the essence of adaptation are identified. The behaviour of a system is formalized in terms of a category of specification-

carrying programs (also called *contracts*), i.e., triples made of a program, a specification and a satisfaction relation among them; arrows between contracts are refinement relations. Contracts are equipped with a functorial semantics, and their adaptive version is obtained by indexing the semantics with respect to a set of *stages of adaptation*, yielding a coalgebraic presentation potentially useful for further generalizations. An adaptation is a transformation of a specification-carrying-program into another one, satisfying some properties. Therefore, the control data includes the entire program being executed.

Different in spirit is our proposal in [19] where we studied the consequences of making a particular choice of control data in automata-like models (and, in particular, in Interface Automata [3], a foundational model of component-based systems). For this purpose we introduced the concept of Adaptable Transition System and its instantiation to Adaptable Interface Automata (AIA), an essential model of adaptive systems inspired by our white-box approach. The key feature of AIAs are control propositions, the formal counterpart of control data. The choice of control propositions is arbitrary, but it imposes a clear separation between ordinary behaviors and adaptive ones.

4.2 Rule-based Models for Adaptation

Rule-based programming is an excellent example of a successful and widely adopted declarative paradigm, thanks to the solid foundations offered by rule-based theoretical frameworks like term and graph rewriting. As many other programming paradigms, several rule-based approaches have been tailored or directly applied to adaptive systems (e.g., graph transformation [33]). Typical solutions include dividing the set of rules into those that correspond to ordinary computations and those that implement adaptation mechanisms, or introducing context-dependent conditions in the rule applications (which essentially corresponds to the use of standard configuration variables). The control data are identified by the above mentioned separation of rules in the first case, and they correspond to the context-dependent conditions in the latter.

The situation is different when we consider rule-based approaches which enjoy higher-order or reflection mechanisms. A good example is *logical reflection*, a key feature of frameworks like rewriting logic [54]. At the ground level, a rewrite theory $\mathcal{R}$ (e.g., a software module) lets us infer a computation step $\mathcal{R} \vdash t \to t'$ from a term (e.g., a program state) t into t'. A universal theory $\mathcal{U}$ lets us infer the computation at the "meta-level", where theories and terms are meta-represented as terms: the above computation step can be expressed in $\mathcal{U}$ as $\mathcal{U} \vdash (\overline{\mathcal{R}}, \overline{t}) \to (\overline{\mathcal{R}}, \overline{t'})$; moreover, the rewrite theory $\mathcal{R}$ can be also rewritten by meta-level rewrite rules, like in $\mathcal{U} \vdash (\overline{\mathcal{R}}, \overline{t}) \to (\overline{\mathcal{R}'}, \overline{t'})$. Since $\mathcal{U}$ itself is a rewrite theory, the reflection mechanism can be iterated yielding what is called the *tower of reflection*, where not only terms $\overline{t}$, but also rewrite rules of the lower level can be accessed and modified at runtime. This mechanism is efficiently supported by Maude [26] and has given rise to many interesting meta-programming applications.

In particular, rewriting logic's reflection has been exploited in [55] to formalize a model for distributed object reflection, suitable for the specification of adaptive systems. Such model, called Reflective Russian Dolls (RRD), has a structure of layered configurations of objects, where each layer can control the execution of objects in the lower layer by accessing and executing their rules, possibly after modifying them, e.g., by injecting some specific adaptation logic in the wrapped components (cf. Fig. 7). The RRD model falls within our conceptual framework by identifying as control data for each layer the rules of its theory that are possibly modified by the upper layer. Note that, while the tower of reflection relies on a white-box architecture, the Russian Dolls approach can deal equally well with black-box components, because wrapped configurations can be managed by message passing. RRD has been further exploited for modeling policy-based coordination [73], for the design of PAGODA, a modular architecture for specifying autonomous systems [74], in the composite actors used in [32], and, by ourselves, in the design and analysis of self-assembly strategies for robot swarms [21].

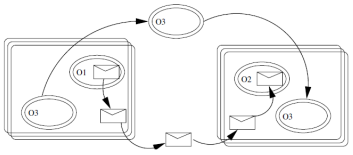

Fig. 7. RRD.

4.3 Concurrency Models for Adaptation

Languages and models conceived in the area of concurrency theory are also good candidates for the specification and analysis of self-adaptive systems. We inspect some paradigmatic formalisms to see how the conceptual framework can help us in the identification of the adaptation logic within each model.

Petri nets are undoubtedly the most popular model of concurrency, based on a set of repositories, called places, and a set of activities, called transitions. The state of a Petri net is called a marking, that is a distribution of resources, called tokens, among the places of the net. A transition is an atomic action that consumes several tokens and produces fresh ones, possibly involving several repositories at once. In *coloured* Petri nets, the tokens can represent structured data and transitions can manipulate them.

The approach proposed in [80] emphasizes the use of Petri nets to validate the development of adaptive systems. Specifically, it represents the local behavioural models with coloured Petri nets, and the adaptation change from one local model to another with an additional Petri net transition labeled `adapt` (cf. Fig. 8). Such `adapt` transitions describe how to transform a state in the source Petri net into a state in the target one, thus providing a clean solution to the *state transfer problem* (i.e., the problem to

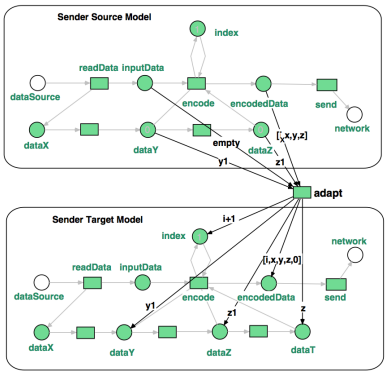

Fig. 8. Adaptive system's Petri net

consistently transfer the state of the system
before and after the adaptation) common to
these approaches. In this context, a natural choice of control data would be the Petri net that describes the current base-level computation, which is replaced during an adaptation by another local model.

Petri nets are used in [62] to formalize multi-layer adaptation in large scale applications spanning over heterogeneous organizations and technologies. Here the multi-layered architecture is motivated by the presence of different languages and technologies addressing their own concerns and views within the same application in a coherent manner and multi-layered adaptation must ensure that coherence between views is always maintained. For example, a three-layers architecture is typical of service-based applications: one layer for service specification (e.g., WSDL); one for behavior description (e.g., BPEL); and one for the organizational view that specifies the stakeholders involved in the business process.

Multi-layer adaptation is triggered by *adaptation events* that are raised by human stakeholders or by layer-specific monitors that discover, e.g., message-ordering mismatches (at the behavior level), or invocation mismatches (at the service layer). Application mismatches are organized along tree-based taxonomies that are put in correspondence with suitable adaptation templates. The main idea is that adaptation techniques that can tackle one application mismatch m can also be used to adapt mismatches that are "below" m in the taxonomy. Cross-layer adaptation is achieved by linking templates at different application layers: templates may trigger the executions of other templates both through direct invocation or by raising other adaptation events. Adaptation templates, the taxonomy navigation and the template-selection environment are modeled as Petri nets (they support the search of the templates starting from the more specific to the more general, w.r.t. the raised adaptation event). As the emphasis is the specification of a generic adaptation model for pervasive applications, the Petri net abstracts away from the execution of multi-layered applications and thus the identification of control data is only possible for concrete instances.

Classical process algebras (CCS, CSP, ACP) are tailored to the modeling of reactive systems and therefore their processes easily fall under the hat of the reactive pattern of adaptation. Instead, characterizing the control data and the adaptation logic is more difficult in this setting. The π-calculus, the join calculus and other nominal calculi, can send and receive channels names, realizing some sort of reflexivity at the level of interaction: they transmit communication media.

An example of use of π-calculus for modeling autonomic systems is [76]. There, adaptive systems are organized in two-levels, local and global. The local level is formed by autonomic elements structured in the MAPE-K spirit as a managed element and an autonomic manager, defined by π-calculus processes that communicate over designated channels. In particular, the effector process enacts adaptation requests by sending messages to its managed element over the effector channel, which acts as the control data (storing a message in the channel triggers adaptation) of the local adaptive behavior. At the global level a centralized autonomic manager monitors and controls the locally distributed

autonomic managers. Again, adaptation is realized by sending messages through suitable effector channels.

Similar approaches have been explored within process calculi that feature primitives adequate to model autonomic systems, including explicit locality aspects, asynchronous communication and code mobility. A paradigmatic example is KLAIM [29], which has been studied as a convenient language for modeling self-adaptive systems in [35]. The authors describe how to adopt in KLAIM three paradigms for adaptation: two that focus on the language-level, namely, context-oriented and aspect-oriented programming (cf. Sections 5.1 and 5.2, respectively), and one that focuses on the architectural-level (MAPE-K).

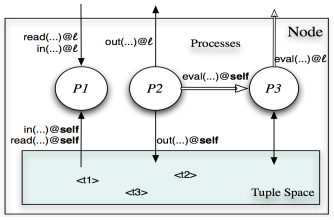

Fig. 9. A KLAIM node.

The main idea is to rely on *process tuples*, that is, tuples (the equivalent of messages in the tuple-space paradigm) that denote entire processes. Process tuples are sent by manager components (locations in KLAIM) to managed components, which can then install them via the *eval* primitive of KLAIM (cf. Fig. 9), i.e., adaptation is achieved by means of code mobility and code injection. The control data in this case amounts to the set of active processes in each location.

Stemming from this approach, the Service Component Ensemble Language (SCEL) has been proposed in [30] which realizes adaptation by combining different paradigms, i.e., policy-based programming (discussed in Section 5.3), tuple-space communication, and knowledge-based reasoning. In this case control data is spread among the policy rules, the process tuples and the knowledge facts.

In [51] the authors present a lightweight approach to service adaptation based on process algebraic techniques. As in [14], adaptation is achieved by the design-time synthesis of service adaptors that act as mediators for the communication between two services and allow to overcome signature and behaviour mismatches between their contracts. Differently from [14], an adaptor process is deployed that is itself adaptive, in the sense that its behaviour is initially distilled on the basis of adaptation contracts and then the adaptor is progressively refined at run-time exploiting the collected information about interaction failures. This is useful when service behavior may evolve at runtime due to changes of the environmental conditions in ways not foreseeable in the contract, e.g., depending on the current load of its server. The approach is lightweight because it introduces low overhead. Learning adaptors have been implemented and included in the Integrated Toolbox for Automatic Composition and Adaptation (ITACA) [25]. The control data of the approach are the adaptors themselves.

We conclude this section by mentioning the approach in [16], where the concept of *adaptable process* has been put forward to model dynamic process evolution patterns in process algebras. Adaptable processes are assigned a location and can be updated at runtime by executing an update prefix related to that location. Roughly, if P is an adaptable process running at location a, written

$a[P]$, and U is a process context, called *update pattern*, then the execution of the update prefix $\tilde{a}\{U\}$ stops the execution of P within a (i.e., $a[P]$ is removed) and replaces it with $U(P)$. Note that location a is not necessarily preserved by the update, providing flexibility on the allowed update capabilities. For example, the prefix $\tilde{a}\{nil\}$ would just remove $a[P]$; the prefix $\tilde{a}\{a[Q]\}$ would replace $a[P]$ by $a[Q]$; the prefix $\tilde{a}\{b[\cdot]\}$ would move P from location a to the location b; and the prefix $\tilde{a}\{a[\cdot|\cdot]\}$ would spawn an extra copy of P within a. The authors exploit the formal model to study undecidability issues of two verification problems, called *bounded* and *eventual adaptation*, i.e., that there is a bound to the number of erroneous states that can be traversed and that whenever a state with errors is entered, then a state without errors will be eventually reached, respectively. The control data of [16] are the adaptable processes of the form $a[P]$.

5 Programming paradigms for Adaptation

As we observed, the nature of control data can vary considerably depending both on the degree of adaptivity of the system and on the nature of the computational formalisms used to implement it. Examples of control data include configuration variables, rules and plans (in rule-based programming), code variations (in context-oriented programming), interactions (in connector-centered approaches), policies (in policy-driven languages), advices (in aspect-oriented languages), monads and effects (in functional languages), and even entire programs (in models of computation exhibiting higher-order or reflective features). Indeed, many programming languages that consider such forms of control data as first-class citizens have been promoted as suitable for programming adaptive systems (see the overviews of [34,70]). Just restricting to Java, technologies supporting adaptation include Jolie [57], ContextJ [8], JavAdaptor [64] and Chameleon [9]. This section surveys a representative set of such programming paradigms and explain their notion of adaptation in terms of CoDa. The approaches are organized according to three paradigms: context-oriented programming (Section 5.1), aspect-oriented programming (Section 5.2), and policy-oriented programming (Section 5.3).

5.1 Context-Oriented Programming for Adaptation

Context-oriented programing [38] (COP) has been designed as a convenient paradigm for programming autonomic systems [69]. The main idea is to rely on a pool of *code variations* choosen according to the program's *context*, i.e., the runtime environment under which the program is running. Under this paradigm the natural choice of control data is the current set of active code variations.

Many languages have been extended to adopt this paradigm. We mention among others Lisp, Python, Ruby, Smalltalk, Scheme, Java, and Erlang. The notion of context varies from approach to approach and it might refer to any computationally accessible information. Without giving any concrete reference, a typical

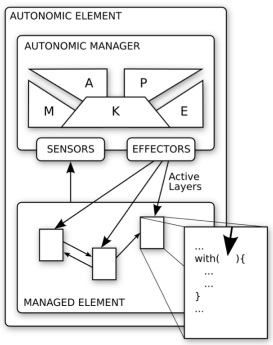

Fig. 10. MAPE-K architecture in COP .

example is the environmental data collected from sensors. In many cases the universe of all possible contexts is discretised in order to have a manageable, abstract set of fixed contexts. This is achieved, for instance by means of functions mapping the environmental data into the set of fixed contexts. Code fragments like methods or functions can then be specialized for each possible context. Such chunks of behaviours associated with contexts are called *variations*.

COP can be used to program autonomic systems by activating or deactivating variations in reaction to context changes. The key mechanism is the dynamic dispatching of variations. When a piece of code is being executed, a dispatcher examines the current context of the execution in order to decide which variation to invoke. Contexts thus act as some sort of possibly nested scopes. Indeed, often a stack is used to store the currently active contexts, and a variation can propagate the invocation to the variation of the enclosing context. The key idea to achieve adaptation along the lines of the MAPE-K framework is for the manager to control the context stack and for the managed component to access it in a read-only manner. The points of the code in which the managed component queries the current context stack are called *activation hooks* (*adaptation hooks* in [48] and in [35], as we shall see in Sections 5.2 and 5.3, respectively).

Given our informal description, COP falls into CoDa assuming the context stack as control data. The only difference between the approach proposed in [69] (cf. Fig. 10) and our ideas is that the former suggests the context stack to reside within the manager (this may not be clear in the figure, and we refer to the example in [69]), while for us the control stack resides in the interface of the managed component, in order to identify such component as an adaptable one.

5.2 Aspect-Oriented Programming for Adaptation

Aspect-oriented programming [45] and, in particular, dynamic aspect-oriented programming [63] have been advocated as a convenient mechanism for developing self-adaptive software by many authors since the original proposal of [36]. The main idea is that the separation-of-concerns philosophy of aspects facilitates the addition of autonomic computing capabilities. Indeed, while early works [36] put the stress on monitoring as an aspect, subsequent works have generalized this idea to other capabilities. Adaptation can be realized through aspect weaving, i.e., the activation and deactivation of advices (the code to be executed at join points), possibly enacted by an autonomic manager. Advices, hence, can be understood as the control data of the aspect-based adaptation paradigm. Dynamic aspect oriented programming languages, equipped with dynamic aspect weaving mechanisms, thus facilitate the realization of dynamic adaptation.

5.3 Policy-Oriented Programming for Adaptation

As we have seen in Section 4.2, rule-based approaches have been advocated as a convenient mechanism for realizing self-adaptation. Another example of this tradition are policies. Generally speaking, policies are in fact rules that determine the behavior of an entity under specific conditions. Policies have been seen as mechanisms enjoying the flexibility required by self-* systems, and tackling the problem at the right (high-) level of abstraction. Quite naturally, adaptation can be realized by changing policies according to the program's current status. The natural choice of control data is then the current set of active policies.

A prominent example is the Policy-based Self-Adaptive Model (PobSAM) [44], a formal framework for modeling and analyzing self-adaptive systems which relies on policies as a high-level mechanism to realize adaptive behaviors. Building upon the authors experience in the development of the PAGODA framework [74] (cf. Section 4.2), PobSAM combines the actor model of coordination [2] with process algebra machinery and shares the white-box spirit of separating application and adaptation concerns. Indeed, the overall architecture of the system is composed by *managed actors*, which implement the functional behavior, and *autonomic manager (meta-)actors*, which control managed actors by enforcing policies. Thus, the adaptation logic is encoded in policies whose responsibility relies on well-identified system components (i.e., the managers). In particular, the configuration of managers is determined by their sets of policies which can vary dynamically. The currently active set of policies represents the control data in this approach. Adaptation is indeed the switch between active policies. Policies are rules that determine under which condition a specified subject must or must not do a certain action. PobSAM distinguishes between *governing* policies, which control the managed actors in their *stable* (cf. steady, normal) state and *adaptation* policies, which drive the actors in the transient states (cf. adaptation phases).

The authors of [48] propose a framework for dynamic adaptation based on the combination of *adaptation hooks*, which specify *where* to apply adaptation, and policies called *adaptation rules*, which specify *when* and *how* to apply it. In their approach an adaptable application is an application that exposes part of its states and the set of activities that it performs in a suitable interface called *application interface*. Adaptation is enacted by suitable managers that exploit the adaptation rules to introduce changes in the application through its interface. In particular, the rules define adaptations that may change the activities by instantiating new code or changing their configuration parameters and may also change part of the application's state. Hence, in this approach both the set of activities and the exposed application state are to be considered as control data in the basic adaptation layer. On top of this basic layer, *dynamic* adaptation can occur, which consists on modifying the adaptation rules at runtime. This makes adaptation managers adaptable as well. At this layer, hence, the control data are the adaptation rules, which determine the behavior of the adaptation managers.

The approach is instantiated in the Java Orchestration Language Interpreter Engine (Jolie) [57], a framework for rapid prototyping of service oriented applica-

tions. The approach is, however, language agnostic, as the authors identify the basic ingredients needed to implement their approach in other settings and a generic architecture to structure it. The former consists of mechanisms needed to implement the adaptation interface and its manipulation based on code mobility. At the architectural level applications are structured as *clients* which rely on an *activity manager* to run their activities. Adaptation is governed by *adaptation servers*, which are coordinated globally by an *adaptation manager service*.

6 Related Work

We have already discussed some of our sources of inspiration in the previous sections and spelled out how their underlying notion of adaptation can be recast in terms of our approach. This section discusses two kinds of related works. Section 6.1 is devoted to works that propose a definition of adaptation. Section 6.2 discusses works that provide a classification of approaches and techniques, guided by a set of dimensions or facets relevant to adaptive systems. Clearly, the references considered here represent only a fragment of the vast literature on adaptive systems: we refer the interested reader to the bibliography of the surveys discussed in this section for completing the picture.

6.1 On the Essence of Adaptation

This sections focuses on other approaches that aim to provide conceptual notions of adaptation. Several proposals follow a black-box perspective that, as discussed in the Introduction, focuses on the external observation of self-adaptive systems.

An interesting contribution is [49], which analyses the notion of adaptation in a general sense and identifies the main concepts around adaptation drawn from different disciplines, including evolution theory, biology, psychology, business, control theory and cybernetics. Furthermore it provides guidelines on the essential features of adaptive systems in order to support their design and understanding.

The author claims that *"in general, adaptation is a process about changing something, so that it would be more suitable or fit for some purpose that it would have not been otherwise"*. The term *adaptability* denotes the capacity of enacting adaptation, and *adaptivity* the degree or extent to which adaptation is enacted. This leads to the identification of four issues that typically play a role in adaptation: context, goals, time-frames, and granularity that are discussed in Section 6.2. The author concludes suggesting that *"due to the relativity of adaptation it does not really matter whether a system is adaptive or not (they all are, in some way or another), but with respect to what it is adaptive"*.

A formal black-box definition is proposed in [17]. If a system reacts differently to the same input stream at different times, then the system is considered to be adaptive, because ordinary systems should exhibit a deterministic behavior. Thus, a non-deterministic reaction is interpreted as an evidence of the fact that the system adapted its behaviour after an interaction with the environment.

Despite its appeal and crispness, we believe that this and similar definitions of adaptation are based on too strong assumptions, restricting considerably its range of applicability. For example, a system where a change of behaviour is triggered by an interaction with the user would not be classified as adaptive.

As we argued in the Introduction, black-box approaches are interesting and useful for evaluating the system robustness under some conditions. However, they are of little use for design purposes where modularization and reuse are critical aspects. Therefore, we believe that a formal definition of adaptation should not be based on the observable behaviour of systems only, as it happens in the black-box approaches. At the same time, we do believe that research efforts are needed to conciliate black-box and white-box perspectives. Ideally, the internal mechanisms and external manifestations of adaptive behavior should be coherent, so that, for instance, a black-box analysis can validate that the degree of adaptability is strongly dependent on the adaptation mechanisms.

A different perspective on adaptation, inspired by the seminal work of IBM on autonomic computing, has been adopted by many authors, e.g., [68]. The starting point is the observation that modern software can be seen as an open loop. Indeed, a software system is inevitably subject to continuous modifications, reparations and maintenance operations which require human intervention. Self-adaptation is seen as the solution to such openness by closing the loop with feedback from the software itself and its context of operation. In this view self-adaptation is seen as a complex feature built upon self-awareness and other self-* mechanisms. Control loops are seen as a fundamental process to achieve adaptive behaviors.

The kind of adaptation discussed so far is concerned essentially with individual components. However it may also happen that a complex system made of non-adaptive components exhibits a collective behavior which is considered to be adaptive (see e.g., the discussion in [49]). Such *emergent adaptation*, typical of massively parallel and distributed systems such as *swarms*, results from the components' interactions. Often, emergent adaptation relies on decentralized coordination mechanisms (e.g., based on the spatial computing paradigm [75,10]). Interesting in this regard can be to shift the focus to *Singerian* forms of adaptation [67,13], where the subject of adaptation is the environment, as opposed to the *Darwinian* one we have focused on, where it is the system who adapts.

A conceptual framework for emergent adaptation would require to shift from a *local* notion of control data to a *global* one, where the control data of the individual components are treated as a whole, possibly requiring mechanisms to amalgamate them for the manager, and to project them back to the components.

6.2 The Facets of Adaptation

The literature on adaptive systems contains several interesting surveys and taxonomies based on the identification of the main facets of adaptation. The concept of control data provides one such facet that has been used in this paper to classify many proposals as discussed in Sections 3–5 and summarized in Fig. 1.

In this section we relate control data with other facets proposed in the literature. In most cases these are orthogonal and provide complementary classification criteria. In a few cases they are closely related with control data, thus providing a more concrete perspective on the corresponding approaches.

The survey on self-adaptive software of [68] is one of the most comprehensive studies on the topic, including also approaches to adaptation from the fields of artificial intelligence, control theory and decision theory. It presents a taxonomy of adaptation concerns, surveys a wide set of representative approaches from many different areas, and identifies some key research challenges. The discussion is driven by the so-called six honest men issues in adaptation: (1) *Why* is adaptation required? Is the purpose of adaptation to meet some robustness criteria, to improve the system's performance or to satisfy some other goal? (2) *When* should adaptation be enacted? Should adaptation be applied reactively or proactively? (3) *Where* is the need to do an adaptation manifested? That is, which artifacts (sensors, variables, etc.) indicate that it is necessary to perform an adaptation? (4) *What* parts of the system should be adapted? That is, which artifacts (variables, components, connectors, interfaces, etc.) have to be modified in order to adapt? (5) *Who* should enact the adaptation? Which entity (e.g., human controller, autonomic manager) is in charge of each adaptation? (6) *How* should adaptation be applied? That is, which is the plan that establishes the order in which to apply the necessary adaptation actions?

Our conceptual framework fits well with this approach and is mainly devoted to the identification of the *where*, which then facilitates finding the right characterization for the remaining *honest men* of a system's adaptation mechanism. In fact, in our view the *where* includes our control data, since it is their manipulation that forces a system to adapt. Interestingly, the taxonomy distinguishes between *weak* adaptation (e.g., modifying parameters) and *strong* adaptation (e.g., replacing entire components): the granularity of control data obviously provides a finer spectrum between these two extremes.

The authors of [52] identify three key technologies that enable the development of adaptive systems and that are nowadays widely accepted: component-based design, separation of concerns, and computational reflection. We remark that our aim is more devoted to providing a common understanding of adaptation rather than promoting particular mechanisms. They argue that there are two main approaches to adaptation: *parameter* adaptation and *compositional* adaptation. In parameter adaptation control data can be identified in those program variables that affect the system behavior, and adaptation coincides with the modification of those variables. Instead, in compositional adaptation control data can be identified in the system's architecture, i.e., in the system components and interconnection, and adaptation coincides with architectural reconfiguration, from replacing entire components to modifying only parts of them. The authors pay a special attention to compositional adaptation and propose a taxonomy that focuses on three main questions: the *when*, *how*, and *where* to compose.

While our aim is centered around the conceptual forms of control data, the authors focus on concrete technological mechanisms and do not consider foundational models such as those discussed in Section 4.

The authors of FORMS (cf. the discussion on [6,77] in Section 3) provide in [5] a classification of modeling facets for self-adaptive systems. The focus is on the underlying conceptual models rather than on the concrete technologies used to realize them. As a result, four main groups of facets are identified: those regarding the *goals* of adaptation, the *changes* that trigger it, the *mechanisms* that realize it, and its *effects*. Goal dimensions include flexibility, duration, and dependency of the system objectives. Change dimensions regard e.g. the source, the frequency, and the level of anticipation of the adaptation triggers. The mechanism-related dimensions range from the type to the level of autonomy, passing through scope, duration, and timeliness. Last, the dimensions concerning the effects of adaptation include criticality, predictability, and resilience. The proposed classes for each facet seem however orthogonal with respect to the choice of control data.

The authors exploit such classes to identify the research challenges of adaptation. They e.g. stress the need of mechanisms to conciliate conflicting goals of participants in open systems; of decentralized mechanisms for coordinating adaptation in distributed systems; and of verification, validation, and prediction mechanisms to ensure that self-adaptive systems behave correctly and predictably.

The survey [15] provides an overview of those approaches that support self-adaptation based on architectural reconfiguration. The authors consider that an architecture is *self-managed* if it can perform architectural changes at runtime by initiating, selecting, and assessing them by itself, without the assistance of an external entity. Contrary to other surveys on architectural reconfiguration (e.g., [27,56]) the focus is on formal models such as graphs, process algebras and logic.

The considered approaches are evaluated in terms of their support for basic reconfigurations such as component or connector addition/removal and composite reconfiguration operations such as sequentialization, iteration and choices. With respect to our proposal, they clearly identify the software achitectures themselves as control data (cf. also the discussion in Section 3.2).

7 Conclusion

We have presented CoDa, a white-box conceptual framework for adaptation that promotes a neat separation of the adaptation logic from the application logic through a clear identification of control data. To validate CoDa we have described a representative set of approaches to (self-)adaptation ranging from architectural solutions (Section 3), to computational models (Section 4), and to programming languages and paradigms (Section 5). For each of them we have highlighted the main distinguishing features and we have discussed the way they fit in CoDa. As a byproduct, our work provides an original perspective from which to survey Computer Science approaches to adaptive systems. We have also discussed (Section 6) other surveys and taxonomies conceived with the aim

to establish a common ground for fruitful research debates by clarifying and identifying the key features of adaptive systems.

The discussion of this paper has also helped us to identify many different forms of control data that can be found in the literature. Our position is that *the* best form of control data does not exist. Every form of control data can be adequate. However, we strongly believe that the choice of control data should adhere to the following three principles (cf. [52]): separation of concerns, component-based design and computational reflection.

Regarding the first two principles, we believe that the choice of control data should neatly separate the application logic from the adaptation logic, and should be clearly identified and encapsulated in a specific component of a suitable adaptation loop, in order to guarantee an understandable, modular design. For this purpose, sound design principles should be developed in order to ensure correctness-by-design, and guidelines for the development of adaptive systems conforming to well-understood patterns.

As for the third principle, we believe that higher-order forms of control data are to be preferred if computationally affordable, since they make it easy to carry the life-cycle of reliable adaptive systems to runtime, by providing runtime models that can be used to monitor, predict and modify the systems.

In Fig. 11 we recap how the (macro) classes of control data identified in Fig. 1 and discussed in Sections 3–5 (i.e., the rows of the table in Fig. 11) have been exploited for adaptation along three pillars of Computer Science (i.e., the columns of the table Fig. 11). Broadly speaking, the presence of blank cells in the table suggests us two main interesting and maybe surprising facts, which are concerned with: (i) the use of reflection in programming languages for adaptation; and (ii) the abstraction from operational aspects in architectural approaches.

While it is out of doubt that reflection offers a natural mechanism to implement adaptation, our analysis shows that it is more common to allow only some controlled form of reflection in languages designed for programming adaptive systems. This is witnessed by the fact that the class "entire program" has no direct representative in the pillar "Languages". Our understanding is that reflection as-it-is does not offer a convenient abstraction to programmers, because it is too powerful and too risky (i.e., error-prone).

Regarding the pillar "Architectures", it seems that the only class of control data exploited for adaptation is that of "architecture" themselves (e.g., components and their connections), whereas operational aspects are disregarded such as those related to the *how* and *why* questions. While one can argue that both classes "entire program" and "operation mode" of adaptation can somehow be represented at the architecture level (e.g., the notion of component replacement can be instantiated to both such classes), we think that the same does not apply to the class "adaptation strategy". This observation was implicit in [15], where a lack in meta-levels for the architectural formalisms was already noted. To fill the gap exposed in Fig. 11, we believe that defining an architectural reference model of adaptation that has adaptation strategies as control data would be an interesting subject of further studies.

	Architectures		Models			Languages		
adaptation strategy			4.1				5.2	5.3
architecture	3.1	3.2	4.1		4.3			5.3
entire program			4.1	4.2	4.3			
operation mode			4.1	4.2	4.3	5.1	5.3	4.3

Fig. 11. Control data classes per pillars

References

1. Adler, R., Schaefer, I., Schüle, T., Vecchié, E.: From model-based design to formal verification of adaptive embedded systems. In: Butler, M., Hinchey, M.G., Larrondo-Petrie, M.M. (eds.) ICFEM 2007. LNCS, vol. 4789, pp. 76–95. Springer (2007)
2. Agha, G.: Actors: a model of concurrent computation in distributed systems. MIT Press (1986)
3. de Alfaro, L., Henzinger, T.A.: Interface automata. In: ESEC/FSE 2001. ACM SIGSOFT Software Engineering Notes, vol. 26(5), pp. 109–120. ACM (2001)
4. Allen, R., Douence, R., Garlan, D.: Specifying and analyzing dynamic software architectures. In: Astesiano, E. (ed.) FASE 1998. LNCS, vol. 1382, pp. 21–37. Springer (1998)
5. Andersson, J., de Lemos, R., Malek, S., Weyns, D.: Modeling dimensions of self-adaptive software systems. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 27–47. Springer (2009)
6. Andersson, J., de Lemos, R., Malek, S., Weyns, D.: Reflecting on self-adaptive software systems. In: SEAMS 2009. pp. 38–47. IEEE Computer Society (2009)
7. Andrade, L.F., Fiadeiro, J.L.: An architectural approach to auto-adaptive systems. In: ICDCSW 2002. pp. 439–444. IEEE Computer Society (2002)
8. Appeltauer, M., Hirschfeld, R., Haupt, M., Masuhara, H.: ContextJ: Context-oriented programming with Java. Journal of the Japan Society for Software Science and Technology on Computer Software 28(1), 272–292 (2011)
9. Autili, M., Benedetto, P.D., Inverardi, P.: A programming model for adaptable java applications. In: Krall, A., Mössenböck, H. (eds.) PPPJ 2010. pp. 119–128. ACM (2010)
10. Beal, J., Cleveland, J., Usbeck, K.: Self-stabilizing robot team formation with proto: Ieee self-adaptive and self-organizing systems 2012 demo entry. In: SASO 2012. pp. 233–234. IEEE Computer Society (2012)
11. Biyani, K.N., Kulkarni, S.S.: Assurance of dynamic adaptation in distributed systems. Journal of Parallel and Distributed Computing 68(8), 1097–1112 (2008)
12. Bosch, J.: Superimposition: a component adaptation technique. Information & Software Technology 41(5), 257–273 (1999)
13. Bouchachia, A., Nedjah, N.: Introduction to the special section on self-adaptive systems: Models and algorithms. ACM Transactions on Autonomous and Adaptive Systems 7(1), 13:1–13:4 (2012)
14. Bracciali, A., Brogi, A., Canal, C.: A formal approach to component adaptation. Journal of Systems and Software 74(1), 45–54 (2005)
15. Bradbury, J.S., Cordy, J.R., Dingel, J., Wermelinger, M.: A survey of self-management in dynamic software architecture specifications. In: Garlan, D., Kramer, J., Wolf, A.L. (eds.) WOSS 2004. pp. 28–33. ACM (2004)

16. Bravetti, M., Giusto, C.D., Pérez, J.A., Zavattaro, G.: Adaptable processes. Logical Methods in Computer Science 8(4), 13:1–13:71 (2012)
17. Broy, M., Leuxner, C., Sitou, W., Spanfelner, B., Winter, S.: Formalizing the notion of adaptive system behavior. In: Shin, S.Y., Ossowski, S. (eds.) SAC 2009. pp. 1029–1033. ACM (2009)
18. Brun, Y., Serugendo, G.D.M., Gacek, C., Giese, H., Kienle, H.M., Litoiu, M., Müller, H.A., Pezzè, M., Shaw, M.: Engineering self-adaptive systems through feedback loops. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 48–70. Springer (2009)
19. Bruni, R., Corradini, A., Gadducci, F., Lafuente, A.L., Vandin, A.: Adaptable transition systems. In: Martí-Oliet, N., Palomino, M. (eds.) WADT 2012. LNCS, vol. 7841, pp. 95–110. Springer (2013)
20. Bruni, R., Corradini, A., Gadducci, F., Lluch-Lafuente, A., Vandin, A.: A conceptual framework for adaptation. In: de Lara, J., Zisman, A. (eds.) FASE 2012. LNCS, vol. 7212, pp. 240–254. Springer (2012)
21. Bruni, R., Corradini, A., Gadducci, F., Lluch-Lafuente, A., Vandin, A.: Modelling and analyzing adaptive self-assembly strategies with Maude. In: Durán, F. (ed.) WRLA 2012. LNCS, vol. 7571, pp. 118–138. Springer (2012)
22. Bucchiarone, A., Cappiello, C., Nitto, E.D., Kazhamiakin, R., Mazza, V., Pistore, M.: Design for adaptation of service-based applications: Main issues and requirements. In: Dan, A., Gittler, F., Toumani, F. (eds.) ICSOC/ServiceWave 2009 Workshops. LNCS, vol. 6275, pp. 467–476. Springer (2010)
23. Bucchiarone, A., Pistore, M., Raik, H., Kazhamiakin, R.: Adaptation of service-based business processes by context-aware replanning. In: Lin, K.J., Huemer, C., Blake, M.B., Benatallah, B. (eds.) SOCA 2011. pp. 1–8. IEEE Computer Society (2011)
24. Cabri, G., Puviani, M., Zambonelli, F.: Towards a taxonomy of adaptive agent-based collaboration patterns for autonomic service ensembles. In: Smari, W.W., Fox, G. (eds.) CTS 2011. pp. 508–515. IEEE Computer Society (2011)
25. Cámara, J., Martín, J.A., Salaün, G., Cubo, J., Ouederni, M., Canal, C., Pimentel, E.: Itaca: An integrated toolbox for the automatic composition and adaptation of web services. In: ICSE 2009. pp. 627–630. IEEE Computer Society (2009)
26. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L.: All About Maude, LNCS, vol. 4350. Springer (2007)
27. Clements, P.: A survey of architecture description languages. In: IWSSD 1996. pp. 16–25. IEEE Computer Society (1996)
28. Cordy, M., Classen, A., Heymans, P., Schobbens, P.Y., Legay, A.: Model checking adaptive software with featured transition systems. In: Cámara, J., de Lemos, R., Ghezzi, C., Lopes, A. (eds.) Assurances for Self-Adaptive Systems, pp. 1–29. Springer (2013)
29. De Nicola, R., Ferrari, G.L., Pugliese, R.: Klaim: A kernel language for agents interaction and mobility. IEEE Transactions on Software Engineering 24(5), 315–330 (1998)
30. De Nicola, R., Loreti, M., Pugliese, R., Tiezzi, F.: A formal approach to autonomic systems programming: The SCEL language. ACM Transactions on Autonomous and Adaptive Systems 9(2), 7:1–7:29 (2014)
31. Dowling, J., Schäfer, T., Cahill, V., Haraszti, P., Redmond, B.: Using reflection to support dynamic adaptation of system software: A case study driven evaluation. In: Cazzola, W., Stroud, R.J., Tisato, F. (eds.) OORaSE 1999. LNCS, vol. 1826, pp. 169–188. Springer (2000)

32. Eckhardt, J., Mühlbauer, T., Meseguer, J., Wirsing, M.: Statistical model-checking for composite actor systems. In: Martí-Oliet, N., Palomino, M. (eds.) WADT 2012. LNCS, vol. 7841, pp. 143–160. Springer (2013)
33. Ehrig, H., Ermel, C., Runge, O., Bucchiarone, A., Pelliccione, P.: Formal analysis and verification of self-healing systems. In: Rosenblum, D.S., Taentzer, G. (eds.) FASE 2010. LNCS, vol. 6013, pp. 139–153. Springer (2010)
34. Ghezzi, C., Pradella, M., Salvaneschi, G.: An evaluation of the adaptation capabilities in programming languages. In: Giese, H., Cheng, B.H.C. (eds.) SEAMS 2011. pp. 50–59. ACM (2011)
35. Gjondrekaj, E., Loreti, M., Pugliese, R., Tiezzi, F.: Modeling adaptation with a tuple-based coordination language. In: Ossowski, S., Lecca, P. (eds.) SAC 2012. pp. 1522–1527. ACM (2012)
36. Greenwood, P., Blair., L.: Using dynamic aspect-oriented programming to implement an autonomic system. In: DAW 2004. pp. 76–88. RIACS (2004)
37. Harvey, I., Paolo, E.A.D., Wood, R., Quinn, M., Tuci, E.: Evolutionary robotics: A new scientific tool for studying cognition. Artificial Life 11(1-2), 79–98 (2005)
38. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented programming. Journal of Object Technology 7(3), 125–151 (2008)
39. Hölzl, M.M., Wirsing, M.: Towards a system model for ensembles. In: Agha, G., Danvy, O., Meseguer, J. (eds.) Formal Modeling: Actors, Open Systems, Biological Systems. LNCS, vol. 7000, pp. 241–261. Springer (2011)
40. Horn, P.: Autonomic computing: IBM's perspective on the state of information technology. IBM (2001)
41. IBM Corporation: An architectural blueprint for autonomic computing. IBM (2005)
42. Iftikhar, M.U., Weyns, D.: A case study on formal verification of self-adaptive behaviors in a decentralized system. In: Kokash, N., Ravara, A. (eds.) FOCLASA 2012. EPTCS, vol. 91, pp. 45–62. EPTCS (2012)
43. Karsai, G., Sztipanovits, J.: A model-based approach to self-adaptive software. Intelligent Systems and their Applications 14(3), 46–53 (1999)
44. Khakpour, N., Jalili, S., Talcott, C., Sirjani, M., Mousavi, M.: Formal modeling of evolving self-adaptive systems. Science of Computer Programming 78(1), 3 – 26 (2012)
45. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J.: Aspect-oriented programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer (1997)
46. Kramer, J., Magee, J.: A rigorous architectural approach to adaptive software engineering. Journal of Computer Science and Technology 24(2), 183–188 (2009)
47. Laddaga, R.: Self-adaptive software: BAA 98-12 proposer information pamphlet. DARPA (1997)
48. Lanese, I., Bucchiarone, A., Montesi, F.: A framework for rule-based dynamic adaptation. In: Wirsing, M., Hofmann, M., Rauschmayer, A. (eds.) TGC 2010. LNCS, vol. 6084, pp. 284–300. Springer (2010)
49. Lints, T.: The essentials in defining adaptation. IEEE Aerospace and Electronic Systems Magazine 1(27), 37–41 (2012)
50. Maraninchi, F., Rémond, Y.: Mode-automata: About modes and states for reactive systems. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 185–199. Springer (1998)
51. Martín, J.A., Brogi, A., Pimentel, E.: Learning from failures: A lightweight approach to run-time behavioural adaptation. In: Arbab, F., Ölveczky, P.C. (eds.) FACS 2011. LNCS, vol. 7253, pp. 259–277. Springer (2012)

52. McKinley, P.K., Sadjadi, S.M., Kasten, E.P., Cheng, B.H.C.: Composing adaptive software. IEEE Computer 37(7), 56–64 (2004)
53. Merelli, E., Paoletti, N., Tesei, L.: A multi-level model for self-adaptive systems. In: Kokash, N., Ravara, A. (eds.) FOCLASA 2012. EPTCS, vol. 91, pp. 112–126. EPTCS (2012)
54. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96(1), 73–155 (1992)
55. Meseguer, J., Talcott, C.: Semantic models for distributed object reflection. In: Magnusson, B. (ed.) ECOOP 2002, LNCS, vol. 2374, pp. 1–36. Springer (2002)
56. Mikic-Rakic, M., Medvidovic, N.: A classification of disconnected operation techniques. In: SEAA 2006. pp. 144–151. IEEE Computer Society (2006)
57. Montesi, F., Guidi, C., Lucchi, R., Zavattaro, G.: JOLIE: a Java orchestration language interpreter engine. In: Boella, G., Dastani, M., Omicini, A., van der Torre, L.W., Cerna, I., Linden, I. (eds.) CoOrg 2006 & MTCoord 2006. ENTCS, vol. 181, pp. 19–33. Elsevier (2007)
58. Mühl, G., Werner, M., Jaeger, M., Herrmann, K., Parzyjegla, H.: On the definitions of self-managing and self-organizing systems. In: KiVS 2007. IEEE Computer Society (2007)
59. O'Grady, R., Groß, R., Christensen, A.L., Dorigo, M.: Self-assembly strategies in a group of autonomous mobile robots. Autonomous Robots 28(4), 439–455 (2010)
60. Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S., Wolf, A.L.: An architecture-based approach to self-adaptive software. Intelligent Systems and their Applications 14(3), 54–62 (1999)
61. Pavlovic, D.: Towards semantics of self-adaptive software. In: Robertson, P., Shrobe, H.E., Laddaga, R. (eds.) IWSAS 2000. LNCS, vol. 1936, pp. 65–74. Springer (2000)
62. Popescu, R., Staikopoulos, A., Brogi, A., Liu, P., Clarke, S.: A formalized, taxonomy-driven approach to cross-layer application adaptation. ACM Transactions on Autonomous and Adaptive Systems 7(1), 7:1–7:30 (2012)
63. Popovici, A., Alonso, G., Gross, T.R.: Just-in-time aspects: efficient dynamic weaving for Java. In: AOSD 2003. pp. 100–109. ACM (2003)
64. Pukall, M., Kästner, C., Cazzola, W., Götz, S., Grebhahn, A., Schröter, R., Saake, G.: Javadaptor - flexible runtime updates of Java applications. Software, Practice and Experience 43(2), 153–185 (2013)
65. Raibulet, C.: Facets of adaptivity. In: Morrison, R., Balasubramaniam, D., Falkner, K.E. (eds.) ECSA 2008. LNCS, vol. 5292, pp. 342–345. Springer (2008)
66. van Renesse, R., Birman, K.P., Hayden, M., Vaysburd, A., Karr, D.A.: Building adaptive systems using ensemble. Software, Practice and Experience 28(9), 963–979 (1998)
67. Sagasti, F.: A conceptual and taxonomic framework for the analysis of adaptive behavior. General Systems XV, 151–160 (1970)
68. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Transactions on Autonomous and Adaptive Systems 4(2), 14:1–14:42 (2009)
69. Salvaneschi, G., Ghezzi, C., Pradella, M.: Context-oriented programming: A programming paradigm for autonomic systems. Tech. Rep. abs/1105.0069, CoRR (2011)
70. Salvaneschi, G., Ghezzi, C., Pradella, M.: Towards language-level support for self-adaptive software. ACM Transactions on Autonomous and Adaptive Systems (2013), to appear

71. Sangiorgi, D.: Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms. Ph.D. thesis, University of Edinburgh (1992)
72. Schaefer, I., Poetzsch-Heffter, A.: Using abstraction in modular verification of synchronous adaptive systems. In: Autexier, S., Merz, S., van der Torre, L.W.N., Wilhelm, R., Wolper, P. (eds.) Trustworthy Software. OASICS, vol. 3. IBFI, Schloss Dagstuhl, Germany (2006)
73. Talcott, C.L.: Coordination models based on a formal model of distributed object reflection. In: Brim, L., Linden, I. (eds.) MTCoord 2005. ENTCS, vol. 150(1), pp. 143–157. Elsevier (2006)
74. Talcott, C.L.: Policy-based coordination in PAGODA: A case study. In: Boella, G., Dastani, M., Omicini, A., van der Torre, L.W., Cerna, I., Linden, I. (eds.) CoOrg 2006 & MTCoord 2006. ENTCS, vol. 181, pp. 97–112. Elsevier (2007)
75. Viroli, M., Casadei, M., Montagna, S., Zambonelli, F.: Spatial coordination of pervasive services through chemical-inspired tuple spaces. ACM Transactions on Autonomous and Adaptive Systems 6(2), 14:1–14:24 (2011)
76. Wang, H., Lv, H., Feng, G.: A self-reflection model for autonomic computing systems based on π-calculus. In: Xiang, Y., Lopez, J., Wang, H., Zhou, W. (eds.) NSS 2009. pp. 310–315. IEEE Computer Society (2009)
77. Weyns, D., Malek, S., Andersson, J.: FORMS: Unifying reference model for formal specification of distributed self-adaptive systems. ACM Transactions on Autonomous and Adaptive Systems 7(1), 8:1–8:61 (2012)
78. Wirth, N.: Algorithms + Data Structures = Programs. Prentice-Hall (1976)
79. Zadeh, L.A.: On the definition of adaptivity. Proceedings of the IEEE 3(51), 469–470 (1963)
80. Zhang, J., Cheng, B.H.C.: Model-based development of dynamically adaptive software. In: Osterweil, L.J., Rombach, H.D., Soffa, M.L. (eds.) ICSE 2006. pp. 371–380. ACM (2006)
81. Zhang, J., Cheng, B.H.C.: Using temporal logic to specify adaptive program semantics. Journal of Systems and Software 79(10), 1361–1369 (2006)
82. Zhang, J., Goldsby, H., Cheng, B.H.C.: Modular verification of dynamically adaptive systems. In: Sullivan, K.J., Moreira, A., Schwanninger, C., Gray, J. (eds.) AOSD 2009. pp. 161–172. ACM (2009)
83. Zhao, Y., Ma, D., Li, J., Li, Z.: Model checking of adaptive programs with mode-extended linear temporal logic. In: EASe 2011. pp. 40–48. IEEE Computer Society (2011)