A Conceptual Framework for Adaptation

Roberto Bruni, Dipartimento di Informatica, Università di Pisa
Andrea Corradini, Dipartimento di Informatica, Università di Pisa
Fabio Gadducci, Dipartimento di Informatica, Università di Pisa
Alberto Lluch Lafuente, IMT Institute for Advanced Studies Lucca
Andrea Vandin, IMT Institute for Advanced Studies Lucca

This paper presents 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 and their role in the adaptation logic. The framework provides an original perspective from which we survey archetypal approaches to (self-)adaptation ranging from programming languages and paradigms, to computational models, to engineering solutions.

Categories and Subject Descriptors: F.1.1 [**Models of computation**]: Self-modifying machines

General Terms: Design, Languages, Theory

Additional Key Words and Phrases: Adaptation, Self-*, Autonomic Computing, Computational Reflection, Control Loops, Context-Oriented Programming

ACM Reference Format:
R. Bruni, A. Corradini, F. Gadducci, A. Lluch Lafuente, A. Vandin 2012. A Conceptual Framework for Adaptation. Draft N/A, N/A, Article N/A (0), 33 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

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 [IBM Corporation 2006].

The literature includes valuable works aimed at capturing the essentials of adaptation both in the most general sense (see e.g. [Lints 2010]) and in particular fields such as that of software systems (see e.g. [Salehie and Tahvildari 2009; Bouchachia and Nedjah 2012; McKinley et al. 2004a; Andersson et al. 2009c; Raibulet 2008]) 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 [Salehie and Tahvildari 2009], 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

This work is supported by the the European Integrated Project 257414 ASCENS.
Author's addresses: R. Bruni, A. Corradini, F. Gadducci: Dipartimento di Informatica, Università di Pisa, Largo Bruno Pontecorvo 3, 56126 Pisa, Italy; A. Lluch Lafuente, A. Vandin: IMT Institute for Advanced Studies Lucca, Piazza San Ponziano 6, 55100 Lucca, Italy.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

self-adaptive systems is carried out in Software Engineering, Artificial Intelligence, Control Theory, and Network and Distributed Computing, among others.

Despite all these classification efforts, there is no agreement on the *conceptual* notion of adaptation, neither in general nor for software systems, not to talk about a widely accepted *foundational* model for it. Lofti Zadeh noticed in [Zadeh 1963] 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. [Raibulet 2008; Salehie and Tahvildari 2009; Andersson et al. 2009c; Lints 2010]), they are valid in Computer Science as well. One of the things that motivates Zadeh's lack of hope for a concrete unifying definition of adaptation is the attempt to subsume two aspects under the same definition: 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* perspective on adaptation, and to the second aspect as the *white-box* one.

Actually, in the realm of Software Engineering there are widely accepted 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"* [Oreizy et al. 1999], where such "environment" or "context" 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"* [Laddaga 1997]. Such definitions can be exploited, to a certain extent, to measure what is often called the *degree of adaptability* or *degree of adaptivity*, i.e. to estimate or predict the system robustness under some conditions. This approach can be traced back to Zadeh's proposal [Zadeh 1963], but has been later adopted by many other authors (e.g. [Mühl et al. 2002; Hölzl and Wirsing 2011]).

The problem is that almost any software system can be considered self-adaptive according to the definitions recalled above, since any realistic system can *modify its behaviour* (for example by following different branches at the same control point) as a *reaction to a change in its context of execution* (like the change of variables or parameters). Therefore such definitions, concerned with the *behavioral* or *observational* perspective only, are of difficult applicability for distinguishing (self-)adaptive systems from plain ("non-adaptive") ones. Furthermore, they are of little use for design purposes, where separation-of-concerns, modularization, reuse and scalability 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 has been several efforts in studying the key primitives for communication, including mechanisms for communicating means to communicate (name mobility) and for code mobility (process passing), which has led to a wide 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 related notions such as *adaptivity* and *adaptability*) are not always useful in pinpointing adaptive systems, even if they allow to discard many systems that certainly are not, and (ii) such definitions do sometimes focus on the issue of *how much* a system adapts to some purpose and less on the issue of *in which manner*.

Contribution. We take a *white-box* perspective that allows us to inspect, to some extent, the internal structure of a system. Our goal is to distill a definition of adaptation that is general enough to be cast for most of the approaches found in the literature, in such a way that our notion of adaptation coincides with that of the authors when instantiated to their work. 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 a concrete answer to basic questions like **"is a software system adaptive?"** or **"where is the adaptation logic in an adaptive system?"**. As we shall explain, our answer is not free of ambiguities, but it enables at least the designer of a system to choose a precise criterion that solves any ambiguity.

Large part of the paper is devoted to a *proof of concept*: we overview several proposals in the literature and validate how our definition of adaptation is applied to them. For example, self-adaptation is often obtained by enriching the software that implements the standard application logic with a control loop which monitors the context of execution, determines the changes to be enforced, and enacts them. Thus systems featuring such an architectural pattern, often called MAPE-K [Horn 2001; IBM Corporation 2006; Kephart and Chess 2003], should definitely be considered as adaptive. But there are other, simpler adaptive patterns, like the Internal Feedback Loop pattern [Cabri et al. 2011], where the control loop is not as neatly separated from the application logic as in MAPE-K, and the Reactive Adaptation pattern, where the system just reacts to events from the environment by changing its behaviour. Also systems realizing such patterns should be captured by a convincing definition of adaptation, and their adaptation logic should be exposed and differentiated from their application logic.

Other software systems that can easily be categorized as (self-)adaptive are those implemented with programming languages either explicitly designed to express these features or explicitly advocated as suitable for that purpose. Archetypal examples are languages belonging to such paradigms as Context Oriented Programming [Hirschfeld et al. 2008; Salvaneschi et al. 2011], where the contexts of execution are first-class citizens, or Dynamic Aspect Oriented Programming [Popovici et al. 2003; Greenwood and Blair. 2004], where dynamic aspect weaving provides a natural mechanism for realizing adaptive behaviors. Nevertheless, it is very important to remark that it is not the programming language what makes a program adaptive or not: truly adaptive systems can be programmed in traditional languages, exactly like object-oriented systems can, with some effort, in traditional imperative languages. However, we stress that, disregarding of the paradigm or language being used, the adaptation logic should be clearly identified and the way it is designed and realized should be well disciplined.

Structure of the paper. The goal of this position paper is to present a conceptual framework for adaptation, proposing a simple structural criterion to portray adaptation (Section 2). To illustrate the framework we have selected a wide representative set of approaches to adaptation. We discuss how systems developed according to them are shown to be adaptive according to our definition and explain how to understand adaptation therein. In particular we organize the overview and discussion of such approaches into three main sections (3–5), respectively devoted to *engineering* aspects (Section 3), *foundational* aspects (Section 4), and *linguistic* aspects (Section 5). Since most of the surveyed approaches cover more than one such aspect, we have decided to consider each work in the section devoted to the aspect that is more stressed in the approach being discussed. In Section 6 we overview other surveys and taxonomies conceived with the same aim as our work: to bring some light around the notion of adaptation, in order to identify the main distinguishing features of self-adaptive systems and establish a common ground for fruitful research debates. Finally, we wrap up our considerations and discuss current and future research in Section 7.

2. WHEN IS A SOFTWARE COMPONENT ADAPTIVE?

Software systems are made of one or more processes, roughly programs in execution, possibly interacting among themselves and with the environment in arbitrarily complex ways. A complex adaptive system, hence, can be built in a compositional way, e.g. by integrating individual adaptive components according to specific patterns as we shall discuss. Indeed, component-based design is a widely accepted key feature of self-adaptive systems [McKinley et al. 2004a]. However, it may also happen that a complex system made of components that are not considered to be adaptive exhibits a collective behavior which is instead considered to be adaptive (see e.g. the discussion in [Lints 2010]). Such *emergent adaptation* is the result of the interaction among components. Our framework focuses on the first form of adaptation, i.e. in the adaptation of (simple or composite) components, for which we introduce the following conceptual framework.

The behavior of a component is governed by a program and according to the traditional view (see e.g. [Wirth 1976]), a program is made of *control* (i.e. algorithms) and *data*. Of course many more sophisticated views and paradigms have been introduced in Computer Science but this very basic view of programs as control and data is sufficient for the sake of introducing our approach. Therefore, we can say that control and data are two conceptual ingredients that in presence of sufficient resources (like computing power, memory or sensors) determine the behaviour of the component. Our conceptual framework of adaptation requires to make explicit the fact that the behaviour of a component depends on some well identified *control data* which can be changed to *adapt* the component's behaviors. At this level of abstraction we are neither concerned with the structure of control data, nor with the way the behaviour of the component is influenced by such data, nor with the causes of modification of control data.

Our definition of adaptation is then very simple but very concrete.

Definition (adaptation). Given a component with a distinguished collection of control data, *adaptation* is the runtime modification of such control data.

From this basic definition we immediately derive several others. A component is *adaptable* if it has a distinguished collection of control data that can be modified at runtime. Thus if either the control data are not identified or they cannot be modified, then the component is not adaptable. Further, a component is *adaptive* if it is adaptable and its control data are actually modified at runtime, at least in some of its executions. Moreover, a component is *self-adaptive* if it modifies its own control data at runtime.

Given the intrinsic complexity of adaptive systems, our conceptual view of adaptation might look like an oversimplification. Our goal is to show that instead it enjoys most of the properties that one would require from such a definition.

Any definition of adaptation should face the problem that the judgement whether a system is adaptive or not is often subjective. Indeed, one can always argue that whatever change in the behaviour the system is able to manifest is part of the application logic, and thus should not be deemed as an adaptation. From our 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
end if
```

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. [O’Grady et al. 2010b]). As some authors observe [Harvey et al. 2005] *“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 just a boolean variable set according to the value returned by some sensors. If `the_hill_is_too_steep` is considered as a standard program variable which is not part of the control data, then the change of behaviour caused by a modification of its value is not considered as an adaptation in our framework. If the variable `the_hill_is_too_steep` is instead considered as part of the control data, then modifications of its value are considered to be adaptations.

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 our white-box perspective 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 and which are not, is passed from the observer of the component to its designer.

We remark that white-box approaches to adaptation have many advantages. For example they allow software engineers to analyze adaptation requirements with the help of the *six honest men* [Salehie and Tahvildari 2009]: (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 adaption be applied reactively or proactively? (3) *Where* should adaptation be enacted? At the business level, at the architectural level, or at some other level? (4) *What* parts of the system should be adapted? That is, which artifacts (components, connectors, interfaces, etc.) should be adapted? (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 follows this spirit and is mainly devoted to the identification of the *what* (i.e. the control data), which then facilitates finding the right *who*, *why*, *when*, *where* and *how* of a system’s adaptation mechanism.

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 for adaptation (those that affect the control data) from those relevant for the application logic only (that should not modify the control data).

Of course, any computational model or programming language can be used to implement an adaptive system, just by identifying the part of the data that governs the behavior. Consequently, the nature of control data can vary considerably, in the range of all possible ways of encapsulating behavior: from simple configuration parameters to a complete representation of the program in execution that can be modified at runtime and passing through aspects, policies, rules, contexts, plans, and all that. The extreme case of having entire program representations as control data is typical of computational models that support meta-programming or reflective features even if, at least in principle, it is possible for any Turing-complete formalism.

We shall discuss in Sections 4 and 5 how adaptation, as defined above, can be obtained in systems implemented according to several computational formalisms and programming paradigms. Before that, we survey in the next section several well accepted architectures, patterns and reference models for adaptive systems and discuss how they can be cast in our framework.

3. ENGINEERING APPROACHES TO ADAPTATION

Several contributions in the literature describe engineering approaches to autonomic computing and self-adaptive software. In this section we survey some of such proposals, stressing for each of them how a reasonable notion of control data can be identified. More explicitly, the criterion that we will 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 subject to a change. We organize the discussion around three main themes: reference models (Section 3.1), architectural approaches (Section 3.2), and model-based approaches (Section 3.3).

3.1. Reference Models for Adaptation

One of the most influential reference models for adaptive systems is the MAPE-K (Monitor, Analyse, Plan, Execute, Knowledge) reference model, introduced in the seminal IBM paper [Horn 2001]. According to it, 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 both the component and the manager implementing the control loop is considered as a self-adaptive component.

This model naturally fits in our framework with an obvious choice for the control data: these are the data of the managed component which are modified by the execute phase of the control loop. Thus the control data of a managed component is (explicitly or implicitly) available through the interface it offers to its manager, which can use it to enact its control loop, as shown in Fig. 1.

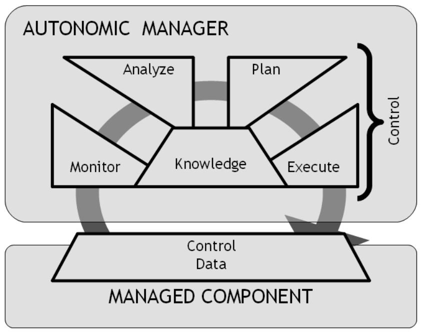


Fig. 1. Control data in MAPE-K.

The construction can be iterated, as the manager itself can be an adaptable component. We shall see several examples of this throughout the paper (e.g. [Lanese et al. 2010; Bucchiarone et al. 2011b]). As an example think of a system like the one in [Bucchiarone et al. 2011b] where 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 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, for instance determining the new strategy 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 itself some control data (conceptually part of its knowledge), thus enabling a hierarchical composition that allows one to build towers of adaptive components (Fig. 2).

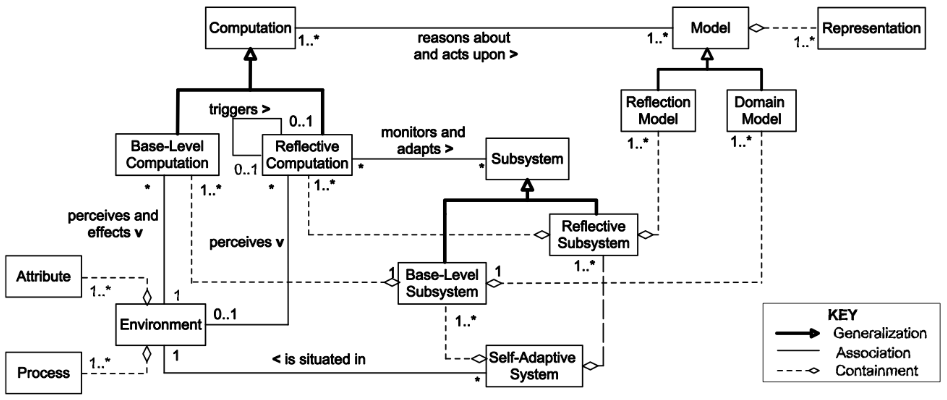


Fig. 3. The FORMS reference model.

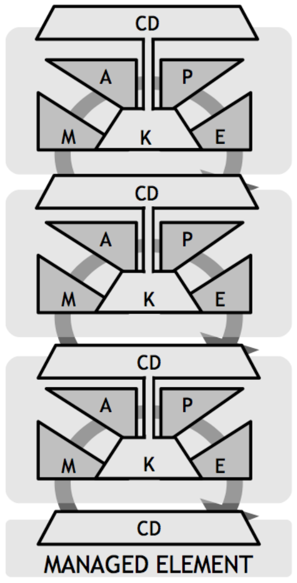


Fig. 2. Tower of adaptation.

Another general reference model has been proposed in [Andersson et al. 2009b], where computational reflection is promoted 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 can inspect and modify the meta-model that is causally connected to the base-level system, so that changes in either one are reflected in the other. The authors argue that most methodologies and frameworks proposed for the design and development of self-adaptive systems rely on some form of reflection, even if this is not always made explicit. Building on these considerations, they introduce the FOrmal Reference Model for Self-adaptation (FORMS) [Weyns et al. 2010; 2012], which provides basic modeling primitives and relationships among them, suitable for the design of self-adaptive systems (cf. Fig. 3). Such primitives allow one to make explicit the presence of reflective (meta-level) subsystems, computations and models.

The goals of [Andersson et al. 2009b] 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 [McKinley et al. 2004a]). The FORMS modeling primitives can be instantiated and composed in a variety of ways. For example, the authors of [Weyns et al. 2010] provide 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 according to the criterion explained above depends on the specific instantiation of the approach, and more precisely on the way modifications to the meta-level (which are adaptations according to the authors) affect the base level. Roughly speaking, control data will be located at the boundary between the meta-level and the base-level components. The same authors provide in [Andersson et al. 2009c] a more general perspective on adaptation that we discuss in detail in Section 6.

In several other reflective systems containing an explicit meta-model of the base-level system, it is pretty clear that such meta-model plays the role of control data. This is precisely the case of systems conforming to the architecture-based solution proposed in [Oreizy et al. 1999]. This approach is concerned with self-adaptation, and considers also issues related to observation, evaluation and planning. It identifies a "spectrum of self-adaptation" which matches pretty well with the spectrum of adaptive systems we have identified: conditional expressions, online algorithms (deterministic, randomized or probabilistic), generic or parametrized algorithms, algorithm selection, evolutionary programming (genetic algorithms, AI-based learning), and so on. Besides the MAPE-K control loop, the approach identifies an architecture-based solution to adaptation: at runtime, an architectural model is mantained that describes the running implementation. This architectural model, made of components and connectors, can be modified by the control loop (adding/removing components/connectors, or changing the topology). An Evolution Manager (EM) ensures the consistency between the architectural model and the implementation, that is modified to reflect the changes to the model.

3.2. Architectural approaches to adaptation

Some of the above discussed reference models put the stress on architectural aspects. Indeed, many architectural solutions to self-* systems have been proposed, mainly based on specific architectures aimed at facilitating the realization of control loops (see e.g. [Brun et al. 2009]), or on architectural reconfiguration as the basic mechanisms for self-adaptation (see the overview of [Bradbury et al. 2004] in Section 6).

A first example is the architectural approach of [Andrade and Fiadeiro 2002]. There, a system specification has a two-layered architecture to enforce a separation between computation and coordination. The first layer includes the basic computational components with well-identified interfaces, while the second one is made of connectors (called *coordination contracts*) that link the components appropriately in order to ensure the required system's functionalities. Adaptation in this context is obtained by *reconfiguration*, which can consist 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 by the verification of certain conditions. In this approach, as adaptation is identified with reconfiguration, the control data consist of the whole two-layered architecture, excluding the internal state of the computational components.

Control loops have been extensively studied (see e.g. the survey of [Brun et al. 2009]) as a key mechanism for achieving self-adaptation in software systems, since their introduction in engineering disciplines like Control Theory. The authors of [Cabri et al. 2011] propose a taxonomy of design patterns for adaptation, which includes typical control loop patterns such as the *internal control loop* and the *external control loop*. Like the MAPE-K reference model, also these patterns can be cast easily in our framework (cf. 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 taxonomy of [Cabri et al. 2011] includes a third pattern called *reactive pattern* that describes *reactive* components (see Fig. 4) capable of modifying their behavior in reaction to an external event, without any control loop. In our conceptual framework, a reactive system of this kind is (self-)adaptive if we consider as control data the variables that are modified by sensing the environment. Other control loops can be found in [Brun et al. 2009] and include, in addition to the MAPE-K control loop, the *Model Identification Adaptive Control loop* and the *Model Reference Adaptive Control loop*.

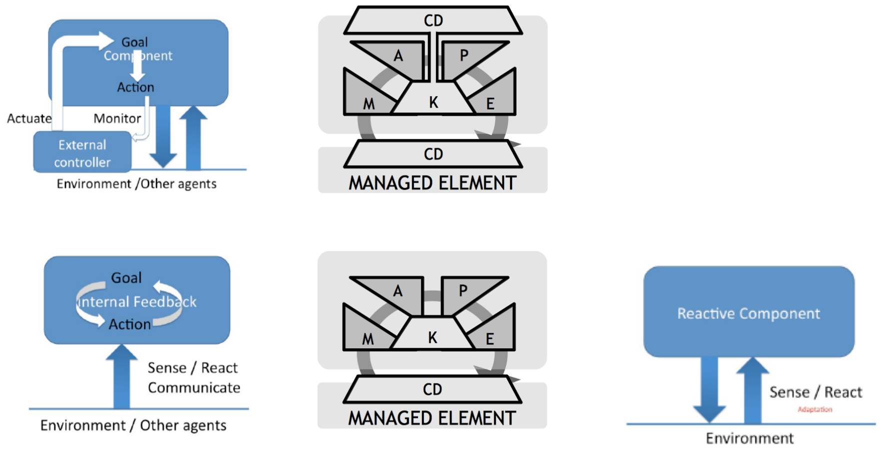


Fig. 4. External (top-left) and internal (bottom-left) control loop patterns and their presentation in terms of the MAPE-K model (center), and the reactive pattern (right).

The Ensemble system [van Renesse et al. 1998] 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. 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 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. Such forms of adaptations are one of the main concerns of the Ensemble system, which are tackled by the *Protocol Switching Protocol* (PSP), 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.

The authors of [Broy et al. 2009] propose a formal definition of when a system exposes an *adaptive behaviour* with respect to a user. A system is modeled as a black-box component that can interact with the user and with the environment through streams of data. An ordinary system is assumed to be deterministic, i.e. it is expected to react in the same way every time the user provides a given input stream. On the other hand, if a system reacts non-deterministically, i.e. by reacting differently to the same input stream provided by the user at different times, the system is considered to be adaptive. The point is that a non-deterministic reaction is interpreted as an evidence of the fact that the system adapted its behaviour after an interaction with the environment. Different kinds of adaptation are considered, depending on how much of the interaction between the environment and the system can be observed by the user. Even if formally crisp, this definition of adaptation is based on strong assumptions, in particular that

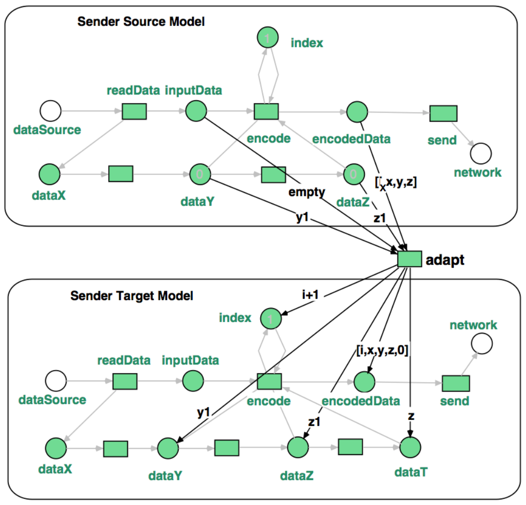


Fig. 5. Petri net model of an adaptive system.

non-adaptive systems are deterministic and that all adaptive systems are interactive. Such assumptions can restrict considerably its range of applicability. For example, it would not classify as adaptive a system where a change of behaviour is triggered by an interaction with the user. More generally, as we argued in the introduction, we think that it is not convenient to base a formal definition of adaptation on a black-box perspective based on the observable behaviour of systems only.

3.3. Model-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, for example for the sake of better performance or to ensure, in case of partial failure, the contractually agreed functionalities, even if in a degraded form. We shall see many examples that fall into this category throughout the paper, in particular in Section 4 and in this section.

A prominent example is the approach proposed in [Zhang and Cheng 2006a], which emphasizes the use of formal methods to validate the development of adaptive systems, for example by requiring the definition of global invariants for the whole system and of local requirements for the "local" behaviours. 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. 5). Such `adapt` transitions describe how to transform a state (a set of tokens) in the source Petri net into a state in the target model, thus providing a clean solution to the *state transfer problem* (i.e. the problem to transfer the state of the system before and after the adaptation in a consistent way) common to these approaches. In this context, a good choice of control data would be the Petri net that describes the current base-level computation, which is replaced during an adaptation change by another local model. Instead, the alternative, pretty natural choice of control data as the tokens that are consumed by the `adapt` transition would be considered poor, as it would not separate clearly the base-level from the meta-level computations.

Also [Karsai and Sztipanovits 1999] shares this view, showing how their approach to Model-Integrated Computing can be applied to adaptive systems. Adaptation is mainly reconfiguration followed by automatic deployment, triggered at runtime either by the user or by the system as reaction to certain events. In the proposed case study, a simple finite state automata determines the transition from one behaviour to another: clearly, the control data of this approach are the states of the finite state automata.

The authors of [Bucchiarone et al. 2010] define a life-cycle for service-based applications where adaptation is a first class concern. Such life-cycle continues during runtime, in order to cope with dynamic requirements and the corresponding adaptations. In addition to the life-cycle, the authors focus 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.

4. COMPUTATIONAL MODELS FOR ADAPTATION

Computational reflection is widely accepted as one of the key instruments to model and build self-adaptive systems (cf. [McKinley et al. 2004a; Dowling et al. 2000]). 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 causing changes of behaviour. Prominent examples of such formalisms are process calculi with higher-order or meta-level aspects (e.g. HO π-calculus [Sangiorgi 1992], MetaKlaim [Ferrari et al. 2004]), higher-order variants of Petri nets (e.g. Dynamic Petri Nets [Asperti and Busi 2009]) and Graph Grammars, rewrite theories with logical reflection like rewriting logic [Meseguer 1992], Logic Programming [Lloyd 1987], and programming languages like LISP, Java, C#, Perl and several others. Systems implemented in these paradigms can realize adaptation within themselves (self-adaptation), but in general the program under execution can be modified also by a different entity, like an autonomic manager implemented in a different language, or in the same language but running in a separate thread. 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) and we argue 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 focus on three main strands: declarative, rule-based computational models (Section 4.1), computational models from the concurrency theory field (Section 4.2), and automata-like computational models (Section 4.3).

4.1. Declarative and Rule-based Models for Adaptation

Logic programming [Lloyd 1987] and its variations are one of the most successful declarative programming paradigms. In the simplest variant, a logic program consists of a set of Horn clauses and, given a goal, a computation proceeds by applying repeatedly SLD-resolution trying to reach the empty clause in order to refuse the initial goal.

Most often logic programming interpreters support two extra-logical predicates, *assert* and *retract*, whose evaluation has the effect of adding or removing the specified Horn clause from the program in execution, respectively, causing a change in its behaviour. This is a pretty natural form of adaptation that fits perfectly in our framework by considering the same clauses of the program as control data. More precisely, this is an example of self-adaptation, because the program itself can modify the control data.

Rule-based programming is another example of a very 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 [Ehrig et al. 2010a]). 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 [Meseguer 1992]. At the ground level, a rewrite theory $\mathcal{R}$ (e.g. a software module) lets us infer a computation step $\mathcal{R} \vdash t \rightarrow t'$ 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: $\mathcal{U} \vdash (\overline{\mathcal{R}}, \overline{t}) \rightarrow (\overline{\mathcal{R}}, \overline{t'})$. 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 rules $\overline{R}$ of the lower level can be accessed and modified at runtime. This mechanism is efficiently supported by Maude [Clavel et al. 2007] and has given rise to many interesting meta-programming applications like analysis and transformation tools.

In particular, the reflection mechanism of rewriting logic has been exploited in [Meseguer and Talcott 2002] to formalize a model for distributed object reflection, suitable for the specification of adaptive systems. Such model, suggestively 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 the rules in their theories, possibly after modifying them, e.g. by injecting some specific adaptation logic in the wrapped components (cf. Fig. 6). Even at this informal level, it is pretty clear that 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. The RRD model has been further exploited, among others, for modeling policy-based coordination [Talcott 2006], for the design of PAGODA, a modular architecture for specifying autonomous systems [Talcott 2007], in the composite actors used in [Eckhardt et al. 2012], and in the design and analysis of self-assembly strategies for robot swarms [Bruni et al. 2012b].

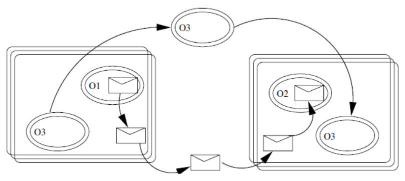


Fig. 6. RRDs.

The RRD approach is also very suitable to model *self-awareness* in software systems, intended as the means by which a software system is *"aware of its self states and behaviors"* [Hinchey and Sterritt 2006]. Indeed, the *self states* can be modeled as the meta-representation of the current state of the objects, while the *self behaviors* can be modeled as the meta-representation of the objects rules.

4.2. Models of Concurrency 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 of the most widely applied 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.

We already discussed an example (namely [Zhang and Cheng 2006a] in Section 3.3) of how Petri nets can be used to model adaptive systems and others (e.g. [Popescu et al. 2012]) can be found in the literature. We provide now a more general discussion which considers different classes of Petri nets and several reasonable choices for control data in addition to the one in [Zhang and Cheng 2006a].

Since the topology of the net is static, there is little room to see a Petri net as an adaptive component: the only possibility is to identify a subset of tokens as control data. Since tokens are typed by repositories, i.e. places, the control data of a Petri net must be a subset CP of its "control" places. Tokens produced or removed from places in CP can enable or inhibit certain activities, i.e. adapt the net. The set CP can then be used to distinguish the adaptation logic from the application logic: if a transition modifies the tokens in CP, then it is part of the adaptation logic, otherwise it regards only the application logic. In particular, the transitions with self-loops on places in CP are those exploiting directly the control data in the application.

Mobile Petri nets [Asperti and Busi 2009] allow the use of colored tokens carrying place names, so that the output places of a transition can depend on the data in the tokens it consumes. In this case, it is natural to include the set of places whose tokens are used as input parameters from some transition in the set of control places. Dynamic nets [Asperti and Busi 2009] are an extension of Mobile Petri Nets that allow for the creation of new subnets when certain transitions fire, so that the topology of the net can grow dynamically. Such "dynamic" transitions (together with the control data individuated for Mobile Petri nets) are natural candidates for the adaptation logic.

Classical process algebras (CCS, CSP, ACP) are certainly 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. Since process algebras are based on message passing facilities over channels, an obvious attempt is to identify suitable adaptation channels. Processes can then be distinguished on the basis of their behavior on such channels, but in general this task is more difficult with respect to Petri nets, because processes will likely mix adaptation, interaction and computation.

The π-calculus [Milner 1999], the join calculus [Fournet and Gonthier 2002] and other nominal calculi, including higher-order versions (e.g. the HO π-calculus [Sangiorgi 1992]) can send and receive channels names, realizing some sort of reflexivity at the level of interaction: they have the ability to communicate transmission media. The situation is then analogous to that of Dynamic Petri nets, as new processes can be spawn in a way which is parametric with respect to the content of the received messages. If again we follow the distinction between adaptation channel names from ordinary channel names, then we inherit all the difficulties described for process algebras and possibly need sophisticated forms of type systems or flow analysis techniques to separate the adaptation logic from the application logic.

An example of the use of the π-calculus for modeling autonomic computing systems can be found in [Wang et al. 2009]. There, adaptive systems are organized in two-levels: the local level and the global one. The local level is formed by autonomic elements structured in the MAPE-K spirit as a managed element and an autonomic manager, all 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 can be understood as the control data of the local adaptive behavior. At the global level a centralized autonomic manager monitors and controls the local 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 that seem adequate to model autonomic systems, including explicit locality aspects, asynchronous communication and code mobility (e.g. based on tuple-spaces). A paradigmatic example is the KLAIM process algebra [De Nicola et al. 1998], which has been studied as a convenient mechanism for modeling self-adaptive systems in [Gjondrekaj et al. 2012]. The authors describe how to adopt in KLAIM three paradigms for adaptation: two that focus on the language-level, namely, context-oriented programming and aspect-oriented programming (that are discussed in Sections 5.1 and 5.2, respectively), and one that focuses on the architectural-level (i.e. MAPE-K).

The main idea in all cases is to rely on the use of *process tuples*, that is, tuples (the equivalent of messages in the tuple-space paradigm) that denote entire processes. These process tuples can be sent by manager components (locations in KLAIM) to managed components, which can then install them via the *eval* primitive of KLAIM (cf. Fig. 7). In other words, 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. Indeed, adaptation in their view is the act of installing a new process.

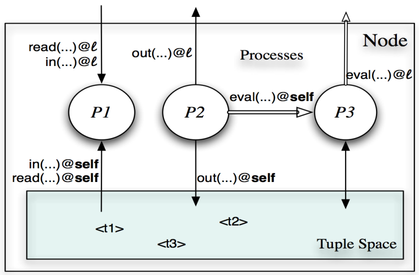


Fig. 7. A KLAIM node.

Stemming from this approach, the *Service Component Ensemble Language* (SCEL) language has been proposed in [De Nicola et al. 2012a] which realizes adaptation by combining different paradigms, namely 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 and clauses.

4.3. Automata-based Approaches to Adaptation

Many of the computational models for adaptation based on automata or transitions systems rely on a multilayered structure reminiscent of hierarchical state machines and automata. In the simple case of two layers, the base layer is formed by models of various functional behaviors of the system, and the upper layer is a model that defines transitions from one base model into another.

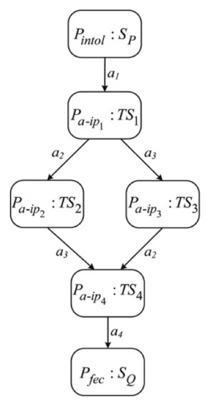
A first example of this tradition are the Adaptive Featured Transition Systems (A-FTS) of [Cordy et al. 2012], which were introduced for the purpose of model checking adaptive software (with a focus on software product lines). A-FTSs are a sort of transition systems where states are composed by the local 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, 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 [Hölzl and Wirsing 2011]). Properties are expressed in AdaCTL, a variant of the computation-tree temporal logic CTL.

Another example of layered computational structures are S[B] systems [Merelli et al. 2012], a model for adaptive systems based on 2-layered transitions systems. The base transition system B defines the ordinary (and adaptable) 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. The paper also introduces formal 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.

As for automata, a prominent example are Mode automata [Maraninchi and Rémond 1998], which have been also advocated as a suitable model for adaptive systems. For example, the approach of [Zhao et al. 2011a] 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*, which implements the adaptation logic and is represented with a mode automaton. Adaptation here is the change of mode (the control data of this approach). The approach considers three different 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 mLTL is used to express such properties.

An automata-based model is also used in [Biyani and Kulkarni 2008] in order to specify and verify correctness of *overlap* adaptations.

Overlap adaptations arise in long-running open and dynamic distributed applications where components can be removed, added or replaced with a certain frequency. The set of components of the application correspond 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.


$P_{intol} : S_P$
a_1
$P_{a\text{-}ip_1} : TS_1$
a_2
a_3
$P_{a\text{-}ip_2} : TS_2$
$P_{a\text{-}ip_3} : TS_3$
a_3
a_2
$P_{a\text{-}ip_4} : TS_4$
a_4
$P_{fec} : S_Q$
Fig. 8. An adaptation lattice.

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* which are, essentially, diamond-shaped graphs whose nodes represent automata and whose transitions correspond to atomic adaptation actions (cf. Fig. 8). 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 adaptation actions.

The lattice imposes further requirements. Notably, the bottom state must be reached at some point (i.e. adaptation cannot be delayed forever). Morever, the lattice-like structure is annotated with invariant properties which express requirements on each of the stages (old, intermediate, new) of the system being adapted.

Actually, the approach considers a finer granularity of components in terms of *fractions*, which are essentially the local instance of components in process locations. This fine-grained granularity introduces a combinatorial explosion in the size of the lattices which has a negative impact in the effort required in their correctness verification. To mitigate this the authors propose a framework based on particular architectures and coordination protocols, where some specialized modules can 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. In this case the control data of the system correspond to such strategies.

Another example of labelled transition system variant used for modeling self-adaptive systems are the *Synchronous Adaptive Systems* (SAS) of MARS [Adler et al. 2007; Schaefer and Poetzsch-Heffter 2006], 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 conditions and the current environment. Control data in this approach are those that determine the active configuration.

A bit different in spirit is the proposal of [Bruni et al. 2013] since the authors investigate a methodology to study the consequences of making a particular choice of control data in an arbitrary model (namely, an Interface Automaton [de Alfaro and Henzinger 2001]). For this purpose they introduce the concept of Adaptable Transition Systems and its instantiation on Adaptable Interface Automata (AIA), an essential model of adaptive systems inspired by the white-box approach to adaptation discussed here, and based on a foundational model of component-based systems. 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, functional behaviors and adaptive ones. The authors discuss how AIAs can be exploited in the specification and analysis of adaptive systems, focusing on various notions proposed in the literature, like adaptability, feedback loops, and control synthesis.

As should be evident, the “programs-of-programs” spirit of most approaches discussed in this section raises scalability and complexity issues. On the other hand, the layered structure of the proposed models can be exploited to study adaptive systems compositionally. As a matter of fact, the authors of [Zhang et al. 2009] 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* [Allen et al. 1998] 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 our framework. In particular, control data here are the individual steady state programs.

This separation of concerns has its counterpart in the property specification language used, namely *Adapt-operator extended LTL* (A-LTL) proposed in [Zhang and Cheng 2006b]. A-LTL extends LTL with an adapt operator that does not provide more expressive power but allows to express properties of adaptive systems in a significantly more concise manner. 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, restriction of adaptations to quiescent states, etc.) in an efficient manner.

Structuring the behavior of adaptive system is a major concern in [Iftikhar and Weyns 2012]. 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). Their work is motivated by the necessity of shifting the focus to behavioral aspects of adaptation, as evidenced in previous experiences of the authors [Weyns et al. 2012] mainly concerned with architectural aspects. In this approach, hence, the control data are those used to characterize the state space zones. The authors use their approach to model and analyze the case study of a decentralized adaptive traffic control system using timed automata and TCTL, 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 [Biyani and Kulkarni 2008]). This idea of specification-carrying-programs is investigated in [Pavlovic 2000]. The author identifies suitable semantical domains aimed at capturing the essence of adaptation. 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. As in many of the above approaches, the control data includes the entire program being executed.

5. PROGRAMMING PARADIGMS FOR ADAPTATION

As observed in the previous sections, 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), contexts (in context-oriented programming), interactions (in connector-centered approaches), policies (in policy-driven languages), aspects (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 [Ghezzi et al. 2011; Salvaneschi et al.]). Just restricting to Java some examples of technologies supporting adaptation include Jolie [Montesi et al. 2007], ContextJ [Appeltauer et al. pear], JavAdaptor [Pukall et al. 2013] and Chameleon [Autili et al. 2010]. We survey in this section a representative set of such programming paradigms and explain their notion of adaptation in terms of our conceptual framework. In particular, we organize the discussed approaches in 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 [Hirschfeld et al. 2008] has been designed as a convenient paradigm for programming autonomic systems [Salvaneschi et al. 2011]. The main idea of this paradigm is that the execution of a program depends on the runtime environment under which the program is running.

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

The context-oriented paradigm can be used to program autonomic systems by activating or deactivating variations in reaction to context changes. The key mechanism exploited here is the dynamic dispatching of variations. When a piece of code is being executed, a sort of 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, very 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.

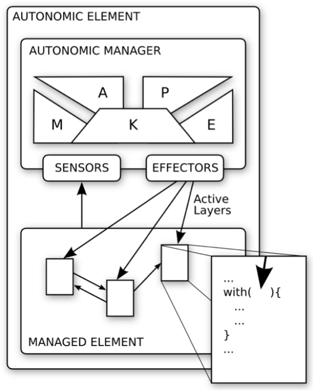


Fig. 9. MAPE-K architecture in context-oriented programming.

The key idea to achieve adaptation along the lines of the MAPE-K framework is for the manager to control the context stack (for example, to modify it in correspondence with environmental changes) and for the managed component to access it in a read-only manner. Those points of the code in which the managed component queries the current context stack are called *activation hooks* (*adaptation hooks* in [Lanese et al. 2010] and in [Gjondrekaj et al. 2012], as we shall see in Sections 5.2 and 5.3, respectively).

Quite naturally, context-oriented programming falls into our framework by considering the context stack as control data. With this view, the only difference between the approach proposed in [Salvaneschi et al. 2011] (cf. Fig. 9) and our ideas is that the former suggests the control data to reside within the manager (this is not clear in the figure but we refer to the detailed example in the cited paper), while we locate the control data in the interface of the managed component.

5.2. Aspect-Oriented Programming for Adaptation

Aspect-oriented programming [Kiczales et al. 1997] and, in particular, dynamic aspect-oriented programming [Popovici et al. 2003] have been advocated as a convenient mechanism for the development of self-adaptive software by many authors since the original proposal of [Greenwood and Blair. 2004].

The main idea is that the separation-of-concerns philosophy of aspects facilitates the addition of autonomic computing capabilities to software systems. Indeed, while early works [Greenwood and Blair. 2004] put the stress on monitoring as an aspect, subsequent works have generalized this idea to other capabilities. Adaptation, for instance, 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, which are equipped with dynamic aspect weaving mechanisms, thus facilitate the realization of dynamic adaptation.

Adaptation hooks are also used in this paradigm. For instance in the tuple-space approach of [Gjondrekaj et al. 2012], already discussed in Section 4.2. The idea is to establish such hooks at join points as checks for adaptations in terms of installation or removal of advices.

5.3. Policy-Oriented Programming for Adaptation

As we have seen in Section 4.1, 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.

A prominent example is PobSAM [Khakpour et al. 2012] (Policy-based Self-Adaptive Model), 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 [Talcott 2007] (cf. Section 4.1), PobSAM combines the actor model of coordination [Agha 1986] 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 of the system, and *autonomic manager (meta-)actors*, which control the behavior of managed actors by enforcing policies. In this manner, 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 [Lanese et al. 2010] 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 in order 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 precisely the adaptation rules, which determine the behavior of the adaptation managers.

The approach is instantiated in Jolie [Montesi et al. 2007] (Java Orchestration Language Interpreter Engine) a framework for rapid prototyping of service oriented applications. The approach is, however, language agnostic. As a matter of fact, the authors identify the basic ingredients needed to implement their approach in other settings and a generic architecture to structure the framework. 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 how the underlying notion of adaptation can be recast in terms of our approach. In this section we focus on other works about adaptation that have inspired our contribution or that have been conceived with a similar aim to ours: conceptual notions of adaptation to explain or capture concrete existing approaches to adaptation. We are aware that they represent only a fragment of the vast literature on adaptive systems which, for obvious reasons, we cannot discuss here in a comprehensive manner. However, we believe that this section covers the most significant surveys on the topic under study, and the references that can be found there can help to complete the picture.

The survey on self-adaptive software of [Salehie and Tahvildari 2009] is one of the most comprehensive studies on the topic. It presents a taxonomy of adaptation concerns, surveys a wide set of representative approaches from many different areas and identifies some key research challenges. All the discussion is driven by the six honest men mentioned in Section 1, namely the *why*, *when*, *where*, *what*, *who*, and *how* issues in adaptation processes. Even if the focus is on self-adaptive software and software engineering, the authors study and discuss also approaches to adaptation from the fields of artificial intelligence, control theory and engineering, decision theory, and network and distributed computing.

In the authors view, modern software can be seen as an open loop, since it has life-cycles that are inevitably subject to continuous modifications, reparations and maintenance operations. Self-adaptive software is 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 other self-* mechanisms (namely self-configuring, self-healing, self-optimizing, and self-protecting), self-awareness and context-awareness. Adaptation loops are seen as a fundamental process to achieve adaptive behaviors. The authors identify an adaptation loop (cf. Fig. 10) reminiscent of that of IBM's MAPE-K reference model, whose main activities are monitoring, detecting, deciding, and acting. Common sensor mechanisms such as logging, profiling and events which feed the monitoring activities are surveyed and so are effector mechanisms like design and architectural patterns, meta-object protocols (cf. Fig. 11) and dynamic aspect weaving, which realize the acting activities.

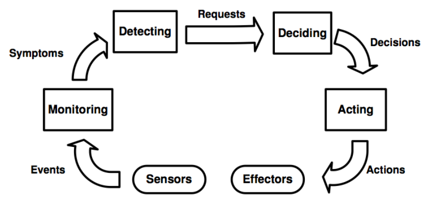


Fig. 10. An adaptation loop.

The taxonomy focuses on four main facets of adaptation: the object of adaptation (*what* and *where*), the mechanisms by which adaptation is realized (*how*), temporal issues (*when*) and interaction concerns (*who*).

Regarding the object to adapt, the taxonomy is refined into issues that concern the structuring of software systems into layers, components and all that. Hence, the *what* and *where* need to consider the layer at which adaptation is applied and the granularity of the artifacts subject to adaptation which may range from single attributes to entire components. The impact and cost of adaptation is another issue which is closely related to the previously mentioned ones. The taxonomy distinguishes between *weak* adaptation (e.g. modifying parameters) and *strong* adaptation (e.g. replacing entire components).

The *how* facet of the taxonomy distinguishes between static (design-time coded) and dynamic (runtime) adaptation, and between external and internal control loops. The authors stress the benefits of dynamic adaptation and the poor scalability and maintainability offered by internal control loops due to the intertwining of the application and the adaptation logic, which contradicts the separation-of-concerns principle that the authors (and many others) promote as key feature of self-adaptive systems. Another issue considered is the distinction between engineering adaptation as designed built-in mechanism and achieving it through generic learning mechanisms, typical of artificial intelligence approaches. Further issues of the *how* facet are whether the system is open or close to runtime alternative adaptations, whether models of the system and the environment are used, or if the adaptation mechanism is generic or domain-specific.

The temporal facet (*when*) focuses on two main issues that dictate the way the decision and monitoring activities are performed within the adaptation loop. Regarding the former, a distinction is made between reactivity and proactivity. Reactive adaptation is triggered by certain events (typically changes in the context or the system itself), while proactive adaptation involves the use of planning or prediction mechanisms to anticipate such events. Monitoring activities can be continuous or adaptive, which means that the features being monitored are dynamically selected, typically trying to restrict them in order to mitigate the introduced overhead.

The last facet regards the interaction between self-adaptive systems and other entities. This involves first the *who* question, the entities the self-adaptive system interacts with. The main distinction is between the fully autonomic view and the human-in-the-loop view, which entails human involvement. Other sub-facets include trust-related issues like security, assurance, dependability, and predictability, and the support for interoperability.

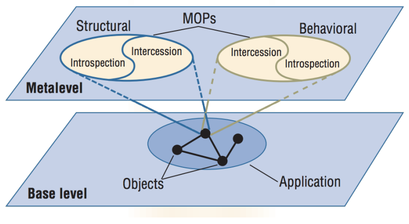


Fig. 11. A reflective system.

The authors of [McKinley et al. 2004a] present a survey of mechanisms to build adaptive systems together with a taxonomy of different forms of compositional adaptation and some key challenges in this field. They argue that there are two main approaches to adaptation: *parameter* adaptation, where program variables that affect the system behavior are modified, and *compositional* adaptation, based on component reconfiguration.

They 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. Our framework promotes all these three technologies, but does not require the latter, although we admit its power and suitability.

The authors pay a special attention to compositional adaptation and propose a taxonomy that focuses on three main questions: the *how*, *when* and *where* to compose. The *how* includes mechanisms such as function pointers, wrappers, proxies, strategies and virtual component patterns, the meta-object protocol (cf. Fig. 11), aspect weaving (cf. Section 5.2), and middlewares (e.g. interception middlewares). The *when* ranges from static composition, carried out at development-, compile/link- and load-time, and dynamic composition, carried out at runtime and which may be *mutable* or *tunable*. In the *where* they discuss middleware-based adaptation versus application-based adaptation.

The authors also discuss some challenges of self-adaptive software: *assurance*, i.e. to ensure the correctness of components with respect to their specifications (cf. the discussion on specification-carrying-programs in Section 4.3) and to guarantee that the system operates safely during adaptation; *security*, i.e. to ensure the integrity of data despite of the invasive mechanisms such as code injection and message interception that may be employed; *interoperability*, i.e. to perform adaptations in a coordinated way in distributed and heterogeneous systems; and *decision making*, i.e. to provide answers to the main questions of the taxonomy, i.e. how, when, and where to adapt.

The authors of FORMS (cf. the discussion on [Andersson et al. 2009b; Weyns et al. 2010; 2012] in Section 3) provide in [Andersson et al. 2009c] a classification of modeling dimensions for self-adaptive systems. Each dimension identifies an aspect that is relevant to self-adaptation in software systems. The main motivation of their work is the lack of agreement in the research and industrial community in what regards those dimensions. Hence, their classification can be used as common ground which provides a simple vocabulary or taxonomy to support the development of new systems or the comparison of existing ones.

The authors have studied several approaches and have focused on the implicit underlying conceptual models rather than on the concrete technologies used to realize them. As a result they identify four main groups of dimensions: dimensions regarding the *goals* or objectives of adaptation, the *changes* that trigger adaptation, the *mechanisms* that realize the adaptation, and the *effects* of adaptation. Goal dimensions include evolution, flexibility, duration, multiplicity and dependency of the system objectives. Change dimensions regard the source, the type, 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 organization, scope, duration, timeliness and triggering. Last, the dimensions that regard the effects of adaptation are criticality, predictability, overhead, and resilience.

The authors identify the research challenges of each of the dimensions. They stress, among other, the need of mechanisms to conciliate conflicting goals in open systems where participants may be in competition; the need of lightweight monitoring and adaptation techniques to mitigate their overhead; the need of decentralized mechanisms for coordinating adaptation in distributed systems; the need of responsive mechanisms in adaptive real-time systems; and the need of verification, validation, and prediction mechanisms to ensure that self-adaptive systems behave correctly and predictably.

The author of [Raibulet 2008] shares our concerns regarding the fact that there are several, apparently different understandings of adaptation, and while she considers the goal of finding a comprehensive definition of adaptation as a difficult, perhaps impossible, task, she proposes to identify the facets of adaptation and the corresponding research challenges.

Different forms of adaptation are identified, namely architectural, structural, behavioral and content-based and the discussion is guided by three main questions: *why is adaptation needed?*, *which are the objectives of adaptation?*, *which are its main open research issues?*. Regarding the first question, the author concludes that, typically, adaptation is needed to cope with resource variability, faults, the complexity of software management (and the inherent overhead), and the convenience to tailor services to users and their contexts. The second question is answered with an enumeration of different fields where adaptation techniques have been employed intensively. These include control engineering systems, operating systems, networks, robotics, artificial intelligence systems, e-learning systems, multimedia applications, information retrieval systems and service-oriented applications such as Web Services.

The author concludes that an adaptive system should be self-aware, context-aware, autonomous and reflective in order to adapt automatically on the basis of meaningful information. In her conclusion the author identifies as necessary research efforts the development of verification and validation techniques to ensure the correctness of self-adaptive systems, and wonders whether issues such as scalability, mobility, late-binding or evolution mechanisms and self-* mechanisms such as self-healing, self-configuration, self-management and self-optimization, are just additional facets of adaptation.

The research roadmap and vision of the Descartes research group described in [Kounev 2011] contains interesting ideas regarding the challenge for engineering the next generation of self-* systems and services. The key mechanism that the authors identify is that of models for *online QoS prediction*. In the MAPE-K terminology this means that the knowledge base contains a model of the performance of the system that can be queried in order to proactively or reactively reconfigure the system.

Actually, they envision an architectural model called the *Online Reconfiguration Process* (cf. Fig. 12) which enriches the MAPE-K reference model with two additional loops: one for refining and calibrating the prediction models, and another one for forecasting the workload evolution.

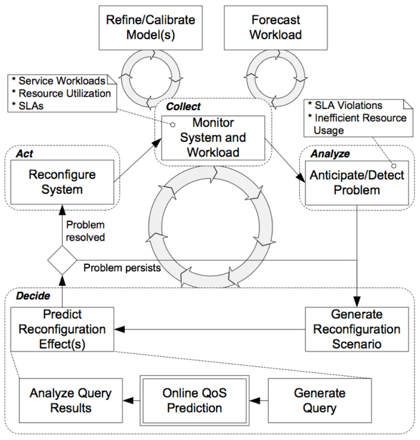


Fig. 12. An online reconfiguration process.

The prediction models are to be exploited at runtime to adapt the system to changes in the environment so to satisfy QoS in near-to-optimal ways, both in terms of utility and resource consumption. For example, self-adaptation actions include the migration of virtual machines in a cloud-based system and the rearrangement of resources in general. The QoS query mechanisms provided by the models are able to predict the effect of such reconfigurations before making a decision and effectively enacting the adaptation.

Notably, self-adaptation is seen as one of the key properties of self-awareness, which may seem to be in contradiction with [Salehie and Tahvildari 2009] where self-awareness is seen as a key mechanism for achieving adaptation. Actually, there is no such contradiction, but rather a different use of the term *self-awareness* which in [Salehie and Tahvildari 2009] and in our opinion should be understood as the ability of a system to be aware of itself (i.e. its own state, architecture, behaviors) while in [Kounev 2011] is used in a more general sense to denote autonomic, self-* systems.

The authors also identify a set of research areas that are contributing (and will continue doing so) to the development of techniques and methods for engineering self-* IT systems. They mention, among others, cloud computing, green computing, service-oriented computing, distributed computing, and event-based computing.

The aim of the survey reported in [Bradbury et al. 2004] is to provide an overview of 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. [Clements 1996; Mikic-Rakic and Medvidovic 2006]) the survey focuses on formal models such as graphs, process algebras and logic.

The surveyed 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. Most of the approaches studied support all such basic reconfigurations and some of the composite ones.

The study puts a special emphasis in the different types of adaptation selection: *predefined*, when the reconfiguration to be applied is predefined at design-time; *constrained* when the reconfiguration to be applied is selected at runtime from a set of possible reconfigurations that was defined at design-time; and *unconstrained*, when there is no restriction on the reconfiguration to be applied.

Another issue considered is the ability of performing distributed reconfigurations in a decentralized way. The authors claim that this is indeed one of the weakness of most of the analyzed approaches, which limits their scalability and hence their ability to cope with large-scale distributed architectures.

The author of [Lints 2010] discusses the notion of adaptation in a very general sense, and identifies the main concepts around adaptation drawn from several different disciplines, including evolution theory, biology, psychology, business, control theory and cybernetics. The main motivation of the author is the vagueness in the use of the term *adaptation*, and the large variety of non-conciliable definitions, which lead to misunderstandings, confusion, and useless debates. The author aims at remedying this by providing some general 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"*. Even if not explicitly stated, the author uses the term *adaptability* to denote the capacity of enacting adaptation, and *adaptivity* for the degree or extent to which adaptation is enacted. This leads to the identification of four main issues that typically play a role in approaches to adaptation: goals, contexts, time-frames, and granularity.

Contexts play a fundamental role in most approaches to adaptation. For instance, in control theory adaptation is a mechanism to deal with external (i.e. contextual) perturbances. Most interestingly, in some fields (e.g. robotics and psychology) the behavior of an entity is considered to be adaptive depending on the context in which such behavior is enacted or even in how regular such context is (cf. the example of this black-box perspective in the field of robotics [Harvey et al. 2005] in Section 2).

Adaptation as an act aimed at fulfilling specific goals or objectives is something common, but not mandatory. There are indeed examples of adaptive systems that are purely reactive and do not have an explicit goal to be pursued. This is often the case in control systems, where *robustness*, the ability to keep the system in some state in spite of external perturbances, is sometimes considered as an adaptation mechanism. Lofti Zadeh [Zadeh 1963], for instance, proposes to consider a system as adaptive with respect to operating conditions and a class of performance values if its performance in those conditions stays within that class. adaptativity in this case is interpreted as a measure to compare adaptive systems on the basis of the set of operating conditions under which a system remains adaptive. Other authors, instead, propose to measure adaptivity as the speed at which the system achieves its goals.

The time frame of adaptation is another of the four main issues. Adaptation, indeed, can regard short- or long-term time frames. Very often a sequence of short-framed actions can be considered as non-adaptive, e.g. worsening the system performance, but they may lead the system toward its goals in the long-term. Long-term adaptation, called *evolution* by some authors, does very often involve evolutionary and learning

CONTROL DATA	ADAPTATION	SECTIONS
Meta-models	Meta-programming and hot-linking	3.1
Architectures	Reconfiguration	3.1, 3.2, 3.3
Entire programs	Meta-programming	4.1
Entire (sub)-programs	Mode switching	3.1, 4.3
Code variations	Variation activation	5.1, 5.2
Policies	Policy enforcement, policy substitution	5.3
Communication means	Communication	4.2
Middleware	Message interception and modification	3.2

Fig. 13. Summary of some the control data forms discussed.

processes. Evolutionary approaches are original from in the field of biology, where adaptation has been often seen as a selective process, which involves the notions of feedback loops and fitness functions, fundamental as well in many other fields, control theory among others. Evolutionary perspectives, however, often assume that adaptation is driven by external selection forces, and adaptation is restricted to variations and selection, but of course many adaptive systems rely on different mechanisms.

Related to the time-framing is the notion of granularity. Adaptation can be seen as a feature of individuals (e.g. survival in biological systems) or collectives (e.g. continuation of the species). As a matter of fact a system, even a software system, may be considered as adaptive even if made of components that are not considered to be adaptive. This is sometimes called *emergent* adaptation.

The author concludes his discussion 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"*.

Other interesting overviews on adaptation include the survey on context-aware service engineering of [Kapitsaki et al. 2009] which focuses on mechanisms to realize context-awareness in adaptive services; the survey on autonomic computing of [Huebscher and McCann 2008], which provides a comprehensive and clear overview of the past, present and future of autonomic computing research; the discussion of [Mühl et al. 2002], which, motivated by the lack of agreement on the meaning of self-managing and self-organizing systems, introduces a classification of such systems building upon Zadeh's definition of adaptive systems; and the work reported in [Fritsch et al. 2008], which describes a classification of automotive software with respect to their adaptation requirements and defines a taxonomy of adaptation dimensions in automotive software.

7. CONCLUSION

We have presented 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 and their role in the adaptation logic. To validate the framework we have described a representative set of archetypal approaches to (self-) adaptation ranging from engineering 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 how they fit in our framework. As a byproduct, our work provides an original perspective from which to survey Computer Science approaches to self-adaptive systems. As a matter of fact, we have also discussed (Section 6) other surveys and taxonomies conceived with the same aim as our work: to establish a common ground for fruitful research debates by clarifying and identifying the key features of self-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. Figure 13 summarizes most of them, together with the main mechanisms used to realize adaptation. Our position is that there does not exist *the* best form of control data. 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. [McKinley et al. 2004a]): 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 (computational reflection), 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 self-adaptive system to runtime, by providing runtime models that can be used to monitor, predict and modify the systems.

Our paper focuses essentially on *designed white-box adaptation*, but there is also a growing interest around the notion of *emerging adaptation* and *black-box adaptation*.

Emergent adaptation is typical of massively parallel and distributed systems such as *swarms* and *ensembles*. A conceptual framework for that kind of systems requires to shift from a *local* notion of control data to a *global* one, where the control data of the individual components of the system are treated as a whole, which will possibly require some mechanism to amalgamate them for the manager, and to project them backwards to the components. Also, such systems use mechanisms coordinate the adaptation of individual components in order to obtain a meaningful adaptation of the whole system. Interesting in this regard can be to shift our focus to *Singerian* forms of adaptation [Sagasti 1970; Bouchachia and Nedjah 2012] where the subject of adaptation is the environment (i.e. control data resides in the environment) - typical of coordination approaches based on the spatial computing paradigm (e.g. [Viroli et al. 2011; Beal et al. 2012]) - as opposed to the *Darwinian* adaptation we have focused on where the system is the subject of the adaptation (i.e. control data resides in the system).

Black-box adaptation focuses on the external observation and requirements of self-adaptive systems, i.e. at measuring or expressing requirements on how a software system adapts his ability to reach some goal under specific context variations. We 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 level of adaptability is strongly dependent on the adaptation logic.

Acknowledgements. This work has been possible thanks to fruitful discussions with many colleagues that motivated and contributed to its development. We would like to thank all of them: the members of the ASCENS project, the organizers and participants of the *AWASS summer school on self-awareness in autonomic computing* held in Edinburgh, and the organizers, participants and anonymous reviewers of the *15th International Conference on Fundamental Approaches to Software Engineering* and the *21st International Workshop on Algebraic Development Techniques* for their valuable criticism and for the opportunity of presenting preliminary versions of this work. In particular, the main differences and novelties with respect to [Bruni et al. 2012a] are (i) the emphasis on the difference between white-box and black-box approaches to adaptation (in particular in Section 1); (ii) the discussion of process algebraic (last part of Section 4.2), automata-based and transition system variants (Section 4.3) specifically designed for adaptive systems; (iii) the discussion of aspect- (Section 5.2) and policy-oriented paradigms (Section 5.3); and (iv) the overview and comparison with respect to similar efforts such as surveys and taxonomies for adaptation (Section 6).

REFERENCES

ADLER, R., SCHAEFER, I., SCHÜLE, T., AND VECCHIÉ, E. 2007. From model-based design to formal verification of adaptive embedded systems. In *Proceedings of the 9th International Conference on Formal Engineering Methods (ICFEM 2007)*, M. Butler, M. G. Hinchey, and M. M. Larrondo-Petrie, Eds. Lecture Notes in Computer Science Series, vol. 4789. Springer, 76–95.

AGHA, G. 1986. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA.

ALDINUCCI, M., DANELUTTO, M., AND VANNESCHI, M. 2006. Autonomic qos in assist grid-aware components. In *Proceedings of the 14th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2006)*. IEEE Computer Society, 221–230.

ALDINUCCI, M. AND TUOSTO, E. 2010. Toward a formal semantics for autonomic components. *CoRR abs/1002.2722*.

ALLEN, R., DOUENCE, R., AND GARLAN, D. 1998. Specifying and analyzing dynamic software architectures. In *(FASE 1998)*. 21–37.

ANDERSSON, J., DE LEMOS, R., MALEK, S., AND WEYNS, D. 2009a. Modeling dimensions of self-adaptive software systems. In *Software Engineering for Self-Adaptive Systems [outcome of a Dagstuhl Seminar]*, B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. Lecture Notes in Computer Science Series, vol. 5525. Springer, 27–47.

ANDERSSON, J., DE LEMOS, R., MALEK, S., AND WEYNS, D. 2009b. Reflecting on self-adaptive software systems. In *Proceedings of the ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2009)*. IEEE Computer Society, 38–47.

ANDERSSON, J., LEMOS, R., MALEK, S., AND WEYNS, D. 2009c. Software engineering for self-adaptive systems. Springer-Verlag, Berlin, Heidelberg, Chapter Modeling Dimensions of Self-Adaptive Software Systems, 27–47.

ANDOVA, S., GROENEWEGEN, L., STAFLEU, J., AND DE VINK, E. P. 2009. Formalizing adaptation on-the-fly. *Electr. Notes Theor. Comput. Sci. 255*, 23–44.

ANDRADE, L. F. AND FIADEIRO, J. L. 2002. An architectural approach to auto-adaptive systems. In *Proceedings of the 22nd International Conference on Distributed Computing Systems Workshops (ICDCSW'02)*. IEEE Computer Society, 439–444.

ANDREI, O. AND KIRCHNER, H. 2009. A higher-order graph calculus for autonomic computing. In *Graph Theory, Computational Intelligence and Thought, Essays Dedicated to Martin Charles Golumbic on the Occasion of His 60th Birthday*, M. Lipshteyn, V. E. Levit, and R. M. McConnell, Eds. Lecture Notes in Computer Science Series, vol. 5420. Springer, 15–26.

APPELTAUER, M., HIRSCHFELD, R., HAUPT, M., AND MASUHARA, H. to appear. Contextj: Context-oriented programming with java. *Journal of the Japan Society for Software Science and Technology (JSSST) on Computer Software*.

ARDAGNA, D., COMUZZI, M., MUSSI, E., PERNICI, B., AND PLEBANI, P. 2007. Paws: A framework for executing adaptive web-service processes. *IEEE Software 24,* 6, 39–46.

ASHLEY-ROLLMAN, M. P., LEE, P., GOLDSTEIN, S. C., PILLAI, P., AND CAMPBELL, J. 2009. A language for large ensembles of independently executing nodes. In *Logic Programming, 25th International Conference, ICLP 2009, Pasadena, CA, USA, July 14-17, 2009. Proceedings*, P. M. Hill and D. S. Warren, Eds. Lecture Notes in Computer Science Series, vol. 5649. Springer, 265–280.

ASPERTI, A. AND BUSI, N. 2009. Mobile petri nets. *Mathematical Structures in Computer Science 19,* 6, 1265–1278.

ASTROM, K. J. AND WITTENMARK, B. 1994. *Adaptive Control* 2nd Ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

AUTILI, M., BENEDETTO, P. D., AND INVERARDI, P. 2009. Context-aware adaptive services: The plastic approach. In *Fundamental Approaches to Software Engineering, 12th International Conference, FASE 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, M. Chechik and M. Wirsing, Eds. Lecture Notes in Computer Science Series, vol. 5503. Springer, 124–139.

AUTILI, M., BENEDETTO, P. D., AND INVERARDI, P. 2010. A programming model for adaptable java applications. In *Proceedings of the 8th International Conference on Principles and Practice of Programming in Java, PPPJ 2010, Vienna, Austria, September 15-17, 2010*, A. Krall and H. Mössenböck, Eds. ACM, 119–128.

BANÂTRE, J.-P., RADENAC, Y., AND FRADET, P. 2004. Chemical specification of autonomic systems. In *Proceedings of the ISCA 13th International Conference on Intelligent and Adaptive Systems and Software Engineering*. ISCA, 72–79.

BARTELS, B. AND KLEINE, M. 2011. A csp-based framework for the specification, verification, and implementation of adaptive systems. In *2011 ICSE Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2011, Waikiki, Honolulu , HI, USA, May 23-24, 2011*, H. Giese and B. H. C. Cheng, Eds. ACM, 158–167.

BASTIDE, G., SERIAI, A., AND OUSSALAH, M. 2007. Software component re-engineering for their runtime structural adaptation. In *Proceedings of the 31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*. IEEE Computer Society, 109–114.

BAUER, S., HENNICKER, R., AND LEGACY, A. 2012. Component interfaces with contracts on ports. In *Proceedings of the 9th International Symposium on Formal Aspects of Component Software (FACS 2012)*.

BEAL, J., CLEVELAND, J., AND USBECK, K. 2012. Self-stabilizing robot team formation with proto: Ieee self-adaptive and self-organizing systems 2012 demo entry. In *Sixth IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2012)*. IEEE Computer Society, 233–234.

BERNARDI, G., BUGLIESI, M., MACEDONIO, D., AND ROSSI, S. 2008. A theory of adaptable contract-based service composition. In *SYNASC 2008, 10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, Timisoara, Romania, 26-29 September 2008*, V. Negru, T. Jebelean, D. Petcu, and D. Zaharie, Eds. IEEE Computer Society, 327–334.

BETTINI, L. AND VENNERI, B. 2011. Object reuse and behavior adaptation in java-like languages. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java (PPPJ 2011)*, C. W. Probst and C. Wimmer, Eds. ACM, 111–120.

BIYANI, K. N. AND KULKARNI, S. S. 2008. Assurance of dynamic adaptation in distributed systems. *J. Parallel Distrib. Comput. 68,* 8, 1097–1112.

BOUCHACHIA, A. AND NEDJAH, N. 2012. 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.

BOUCHENAK, S., BOYER, F., CLAUDEL, B., DE PALMA, N., GRUBER, O., AND SICARD, S. 2011. From autonomic to self-self behaviors: The jade experience. *ACM Transactions on Autonomous and Adaptive Systems 6,* 4, 28:1–28:22.

BRADBURY, J. S., CORDY, J. R., DINGEL, J., AND WERMELINGER, M. 2004. A survey of self-management in dynamic software architecture specifications. In *Proceedings of the 1st ACM SIGSOFT Workshop on Self-Managed Systems (WOSS 2004)*, D. Garlan, J. Kramer, and A. L. Wolf, Eds. ACM, 28–33.

BROGI, A., CANAL, C., AND PIMENTEL, E. 2004. Behavioural types and component adaptation. In *Algebraic Methodology and Software Technology, 10th International Conference, AMAST 2004, Stirling, Scotland, UK, July 12-16, 2004, Proceedings*, C. Rattray, S. Maharaj, and C. Shankland, Eds. Lecture Notes in Computer Science Series, vol. 3116. Springer, 42–56.

BROY, M., LEUXNER, C., SITOU, W., SPANFELNER, B., AND WINTER, S. 2009. Formalizing the notion of adaptive system behavior. In *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC 2009)*, S. Y. Shin and S. Ossowski, Eds. ACM, 1029–1033.

BRUN, Y., SERUGENDO, G. D. M., GACEK, C., GIESE, H., KIENLE, H. M., LITOIU, M., MÜLLER, H. A., PEZZÈ, M., AND SHAW, M. 2009. Engineering self-adaptive systems through feedback loops. In *Software Engineering for Self-Adaptive Systems [outcome of a Dagstuhl Seminar]*, B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. Lecture Notes in Computer Science Series, vol. 5525. Springer, 48–70.

BRUNI, R., CORRADINI, A., GADDUCCI, F., LAFUENTE, A. L., AND VANDIN, A. 2013. Adaptable transition systems. In *Proceedings of the 21st International Workshop on Algebraic Development Techniques (WADT 2012)*. Number 7841.

BRUNI, R., CORRADINI, A., GADDUCCI, F., LLUCH-LAFUENTE, A., AND VANDIN, A. 2012a. A conceptual framework for adaptation. In *Proceedings of the Fundamental Approaches to Software Engineering - 15th International Conference (FASE 2012)*, J. de Lara and A. Zisman, Eds. Lecture Notes in Computer Science Series, vol. 7212. Springer, 240–254.

BRUNI, R., CORRADINI, A., GADDUCCI, F., LLUCH-LAFUENTE, A., AND VANDIN, A. 2012b. Modelling and analyzing adaptive self-assembly strategies with Maude. In *Proceedings of the 9th International Workshop on Rewriting Logic and Its Applications (WRLA 2012)*, F. Durán, Ed. Lecture Notes in Computer Science Series, vol. 7571. Springer, 118–138.

BUCCHIARONE, A., CAPPIELLO, C., NITTO, E. D., KAZHAMIAKIN, R., MAZZA, V., AND PISTORE, M. 2010. Design for adaptation of service-based applications: Main issues and requirements. In *Proceedings of the 2009 international conference on Service-oriented computin (ICSOC/ServiceWave 2009)*, A. Dan, F. Gittler, and F. Toumani, Eds. LNCS Series, vol. 6275. 467–476.

BUCCHIARONE, A., MARCONI, A., PISTORE, M., AND SIRBU, A. 2011a. A context-aware framework for business processes evolution. In *Workshops Proceedings of the 15th IEEE International Enterprise*

Distributed Object Computing Conference, EDOCW 2011, Helsinki, Finland, August 29 - September 2, 2011. IEEE Computer Society, 146–154.

BUCCHIARONE, A., PISTORE, M., RAIK, H., AND KAZHAMIAKIN, R. 2011b. Adaptation of service-based business processes by context-aware replanning. In *2011 IEEE International Conference on Service-Oriented Computing and Applications (SOCA 2011)*, K.-J. Lin, C. Huemer, M. B. Blake, and B. Benatallah, Eds. IEEE, 1–8.

BUCKLEY, J., MENS, T., ZENGER, M., RASHID, A., AND KNIESEL, G. 2005. Towards a taxonomy of software change. *Journal of Software Maintenance 17,* 5, 309–332.

CABRI, G., PUVIANI, M., AND ZAMBONELLI, F. 2011. Towards a taxonomy of adaptive agent-based collaboration patterns for autonomic service ensembles. In *CTS 2011*, W. W. Smari and G. C. Fox, Eds. IEEE Computer Society, 508–515.

CERNUZZI, L., MOLESINI, A., OMICINI, A., AND ZAMBONELLI, F. 2011. Adaptable multi-agent systems: The case of the Gaia methodology. *International Journal of Software Engineering and Knowledge Engineering 21,* 4, 491–521.

CHEN, W.-K., HILTUNEN, M. A., AND SCHLICHTING, R. D. 2001. Constructing adaptive software in distributed systems. In *ICDCS*. 635–643.

CHENG, B. H. C., DE LEMOS, R., GIESE, H., INVERARDI, P., MAGEE, J., ANDERSSON, J., BECKER, B., BENCOMO, N., BRUN, Y., CUKIC, B., SERUGENDO, G. D. M., DUSTDAR, S., FINKELSTEIN, A., GACEK, C., GEIHS, K., GRASSI, V., KARSAI, G., KIENLE, H. M., KRAMER, J., LITOIU, M., MALEK, S., MIRANDOLA, R., MÜLLER, H. A., PARK, S., SHAW, M., TICHY, M., TIVOLI, M., WEYNS, D., AND WHITTLE, J. 2009. Software engineering for self-adaptive systems: A research roadmap. In *Software Engineering for Self-Adaptive Systems [outcome of a Dagstuhl Seminar]*, B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. Lecture Notes in Computer Science Series, vol. 5525. Springer, 1–26.

CHENG, S.-W. AND GARLAN, D. 2012. Stitch: A language for architecture-based self-adaptation. *Journal of Systems and Software 85,* 12, 2860–2875.

CHOI, O. AND YOON, Y. 2007. A meta data model of context information for dynamic service adaptation on user centric environment. In *MUE*. IEEE Computer Society, 108–113.

CLAVEL, M., DURÁN, F., EKER, S., LINCOLN, P., MARTÍ-OLIET, N., MESEGUER, J., AND TALCOTT, C. L. 2007. *All About Maude*. LNCS Series, vol. 4350. Springer.

CLEMENTS, P. 1996. A survey of architecture description languages. In *International workshop on software specification and design*. IEEE Computer Society Press, 16–25.

CORDY, M., CLASSEN, A., HEYMANS, P., SCHOBBENS, P.-Y., AND LEGAY, A. 2012. Model checking adaptive software with featured transition systems. In *Proceedings of the 4th Workshop on Games for Design, Verification and Synthesis*.

DAVID, P.-C. AND LEDOUX, T. 2003. Towards a framework for self-adaptive component-based applications. J.-B. Stefani, I. M. Demeure, and D. Hagimont, Eds. Lecture Notes in Computer Science Series, vol. 2893. Springer, 1–14.

DE ALFARO, L. AND HENZINGER, T. A. 2001. Interface automata. In *ESEC / SIGSOFT FSE*. 109–120.

DE LEMOS, R., GIESE, H., MÜLLER, H., SHAW, M., ANDERSSON, J., BARESI, L., BECKER, B., BENCOMO, N., BRUN, Y., CIKIC, B., DESMARAIS, R., DUSTDAR, S., ENGELS, G., GEIHS, K., GOESCHKA, K. M., GORLA, A., GRASSI, V., INVERARDI, P., KARSAI, G., KRAMER, J., LITOIU, M., LOPES, A., MAGEE, J., MALEK, S., MANKOVSKII, S., MIRANDOLA, R., MYLOPOULOS, J., NIERSTRASZ, O., PEZZÈ, M., PREHOFER, C., SCHÄFER, W., SCHLICHTING, W., SCHMERL, B., SMITH, D. B., SOUSA, J. P., TAMURA, G., TAHVILDARI, L., VILLEGAS, N. M., VOGEL, T., WEYNS, D., WONG, K., AND WUTTKE, J. 2011. Software Engineering for Self-Adpaptive Systems: A second Research Roadmap. In *Software Engineering for Self-Adaptive Systems*, R. de Lemos, H. Giese, H. Müller, and M. Shaw, Eds. Number 10431 in Dagstuhl Seminar Proceedings. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, Dagstuhl, Germany.

DE NICOLA, R., FERRARI, G., LORETI, M., AND PUGLIESE, R. 2012a. A Language-based Approach to Autonomic Computing. In *FMCO 2011*. LNCS 7542. Springer, 25–48.

DE NICOLA, R., FERRARI, G. L., AND PUGLIESE, R. 1998. Klaim: A kernel language for agents interaction and mobility. *IEEE Trans. Software Eng. 24,* 5, 315–330.

DE NICOLA, R., LORETI, M., PUGLIESE, R., AND TIEZZI, F. 2012b. SCEL: a Language for Autonomic Computing. Technical Report. `http://rap.dsi.unifi.it/scel/`.

DONG, X., HARIRI, S., XUE, L., CHEN, H., ZHANG, M., PAVULURI, S., AND RAO, S. 2003. Autonomia: an autonomic computing environment. In *Performance, Computing, and Communications Conference, 2003. Conference Proceedings of the 2003 IEEE International*. 61 – 68.

DOWLING, J., SCHÄFER, T., CAHILL, V., HARASZTI, P., AND REDMOND, B. 2000. Using reflection to support dynamic adaptation of system software: A case study driven evaluation. In *Proceedings of the 1st OOPSLA*

Workshop on Reflection and Software Engineering, W. Cazzola, R. J. Stroud, and F. Tisato, Eds. Lecture Notes in Computer Science Series, vol. 1826. Springer, 169–188.

ECKHARDT, J., MÜHLBAUER, T., MESEGUER, J., AND WIRSING, M. 2012. Statistical model-checking for composite actor systems. In *Preproceedings of the 21st International Workshop on Algebraic Development Techniques (WADT 2012)*.

EHRIG, H., ERMEL, C., RUNGE, O., BUCCHIARONE, A., AND PELLICCIONE, P. 2010a. Formal analysis and verification of self-healing systems. In *FASE 2010*, D. Rosenblum and G. Taentzer, Eds. LNCS Series, vol. 6013. Springer, 139–153.

EHRIG, H., ERMEL, C., RUNGE, O., BUCCHIARONE, A., AND PELLICCIONE, P. 2010b. Formal analysis and verification of self-healing systems. In *Proceedings of the 13th International Conference on Fundamental Approaches to Software Engineering (FASE 2010)*, D. S. Rosenblum and G. Taentzer, Eds. Lecture Notes in Computer Science Series, vol. 6013. Springer, 139–153.

FERRARI, G. L., MOGGI, E., AND PUGLIESE, R. 2004. Metaklaim: a type safe multi-stage language for global computing. *Mathematical Structures in Computer Science 14,* 3, 367–395.

FOURNET, C. AND GONTHIER, G. 2002. The join calculus: A language for distributed mobile programming. In *International Summer School on Applied Semantics (APPSEM 2000)*, G. Barthe, P. Dybjer, L. Pinto, and J. Saraiva, Eds. Lecture Notes in Computer Science Series, vol. 2395. Springer, 268–332.

FRITSCH, S., SENART, A., SCHMIDT, D. C., AND CLARKE, S. 2008. Time-bounded adaptation for automotive system software. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, W. Schäfer, M. B. Dwyer, and V. Gruhn, Eds. ACM, 571–580.

GARLAN, D., CHENG, S.-W., HUANG, A.-C., SCHMERL, B. R., AND STEENKISTE, P. 2004. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer 37,* 10, 46–54.

GEORGAS, J. C., VAN DER HOEK, A., AND TAYLOR, R. N. 2009. Using architectural models to manage and visualize runtime adaptation. *IEEE Computer 42,* 10, 52–60.

GHEZZI, C. 2012. Evolution, adaptation, and the quest for incrementality. In *Large-Scale Complex IT Systems. Development, Operation and Management - 17th Monterey Workshop 2012, Oxford, UK, March 19-21, 2012, Revised Selected Papers*, R. Calinescu and D. Garlan, Eds. Lecture Notes in Computer Science Series, vol. 7539. Springer, 369–379.

GHEZZI, C., PRADELLA, M., AND SALVANESCHI, G. 2011. An evaluation of the adaptation capabilities in programming languages. In *SEAMS 2011*, H. Giese and B. H. Cheng, Eds. ACM, 50–59.

GJONDREKAJ, E., LORETI, M., PUGLIESE, R., AND TIEZZI, F. 2012. Modeling adaptation with a tuple-based coordination language. In *Proceedings of the ACM Symposium on Applied Computing (SAC 2012)*, S. Ossowski and P. Lecca, Eds. ACM, 1522–1527.

GORTON, I., LIU, Y., AND TRIVEDI, N. 2008. An extensible and lightweight architecture for adaptive server applications. *Softw., Pract. Exper. 38,* 8, 853–883.

GREENWOOD, P. AND BLAIR., L. 2004. Using dynamic aspect-oriented programming to implement an autonomic system. In *Proceedings of the 2004 Dynamic Aspects Workshop (DAW04)*. RIACS, 76–88.

GREENWOOD, P. AND BLAIR, L. 2006. A framework for policy driven auto-adaptive systems using dynamic framed aspects. *4242*, 30–65.

GROENEWEGEN, L. AND DE VINK, E. P. 2006. Evolution on-the-fly with paradigm. In *Proceedings of the 8th International Conference on Coordination Models and Languages (COORDINATION 2006)*, P. Ciancarini and H. Wiklicky, Eds. Lecture Notes in Computer Science Series, vol. 4038. Springer, 97–112.

GÜDEMANN, M., ORTMEIER, F., AND REIF, W. 2006. Formal modeling and verification of systems with self-x properties. In *Autonomic and Trusted Computing, Third International Conference, ATC 2006, Wuhan, China, September 3-6, 2006, Proceedings*, L. T. Yang, H. Jin, J. Ma, and T. Ungerer, Eds. Lecture Notes in Computer Science Series, vol. 4158. Springer, 38–47.

HARVEY, I., PAOLO, E. A. D., WOOD, R., QUINN, M., AND TUCI, E. 2005. Evolutionary robotics: A new scientific tool for studying cognition. *Artificial Life 11,* 1-2, 79–98.

HE, J., GAO, T., HAO, W., YEN, I.-L., AND BASTANI, F. B. 2007. A flexible content adaptation system using a rule-based approach. *IEEE Trans. Knowl. Data Eng. 19,* 1, 127–140.

HINCHEY, M. G. AND STERRITT, R. 2006. Self-managing software. *IEEE Computer 39,* 2, 107–109.

HIRSCHFELD, R., COSTANZA, P., AND NIERSTRASZ, O. 2008. Context-oriented programming. *Journal of Object Technology 7,* 3, 125–151.

HÖLZL, M. M. AND WIRSING, M. 2011. Towards a system model for ensembles. In *Formal Modeling: Actors, Open Systems, Biological Systems - Essays Dedicated to Carolyn Talcott on the Occasion of Her 70th Birthday*, G. Agha, O. Danvy, and J. Meseguer, Eds. LNCS Series, vol. 7000. Springer, 241–261.

HORN, P. 2001. Autonomic Computing: IBM's Perspective on the State of Information Technology.

HUEBSCHER, M. C. AND MCCANN, J. A. 2008. A survey of autonomic computing - degrees, models, and applications. *ACM Comput. Surv. 40,* 3.

IBM CORPORATION. 2006. *An Architectural Blueprint for Autonomic Computing*.

IFTIKHAR, U. AND WEYNS, D. 2012. A case study on formal verification of self-adaptive behaviors in a decentralized system. In *Proceedings of the 11th International Workshop on Foundations of Coordination Languages and Self Adaptation (FOCLASA 2012)*.

KAPITSAKI, G. M., PREZERAKOS, G. N., TSELIKAS, N. D., AND VENIERIS, I. S. 2009. Context-aware service engineering: A survey. *Journal of Systems and Software 82,* 8, 1285–1297.

KARSAI, G. AND SZTIPANOVITS, J. 1999. A model-based approach to self-adaptive software. *Intelligent Systems and their Applications 14,* 3, 46–53.

KELLER, R. AND HÖLZLE, U. 1998. Binary component adaptation. In *Proceedings of the 12th European Conference on - Object-Oriented Programming (ECOOP'98)*, E. Jul, Ed. Lecture Notes in Computer Science Series, vol. 1445. Springer, 307–329.

KEPHART, J. O. AND CHESS, D. M. 2003. The vision of autonomic computing. *Computer 36,* 1, 41–50.

KHAKPOUR, N., JALILI, S., TALCOTT, C., SIRJANI, M., AND MOUSAVI, M. 2012. Formal modeling of evolving self-adaptive systems. *Science of Computer Programming 78,* 1, 3 – 26. Special Section: Formal Aspects of Component Software (FACS09).

KHAKPOUR, N., KHOSRAVI, R., SIRJANI, M., AND JALILI, S. 2010. Formal analysis of policy-based self-adaptive systems. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC), Sierre, Switzerland, March 22-26, 2010*, S. Y. Shin, S. Ossowski, M. Schumacher, M. J. Palakal, and C.-C. Hung, Eds. ACM, 2536–2543.

KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C., MARC LOINGTIER, J., AND IRWIN, J. 1997. Aspect-oriented programming. In *ECOOP*. SpringerVerlag.

KICZALES, G. AND RIVIERES, J. D. 1991. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA.

KOUNEV, S. 2011. Self-Aware Software and Systems Engineering: A Vision and Research Roadmap. In *GI Softwaretechnik-Trends, 31(4), November 2011, ISSN 0720-8928*. Karlsruhe, Germany.

KULKARNI, S. S. AND BIYANI, K. N. 2004. Correctness of component-based adaptation. In *Component-Based Software Engineering, 7th International Symposium, CBSE 2004, Edinburgh, UK, May 24-25, 2004, Proceedings*, I. Crnkovic, J. A. Stafford, H. W. Schmidt, and K. C. Wallnau, Eds. Lecture Notes in Computer Science Series, vol. 3054. Springer, 48–58.

LADDAGA, R. 1997. Self-adaptive software. Tech. rep. Tech. Rep. 98-12, DARPA BAA.

LANESE, I., BUCCHIARONE, A., AND MONTESI, F. 2010. A framework for rule-based dynamic adaptation. In *Proceedings of the 5th international conference on Trustworthly global computing*. TGC'10. Springer.

LI, Z. AND PARASHAR, M. 2005. Rudder: An agent-based infrastructure for autonomic composition of grid applications. *Multiagent and Grid Systems 1,* 3, 183–195.

LINTS, T. 2010. The essentials in defining adaptation. In *Proceedings of the 4th Annual IEEE Systems Conference*. 113–116.

LLOYD, J. W. 1987. *Foundations of Logic Programming, 2nd Edition*. Springer.

LYMBEROPOULOS, L., LUPU, E., AND SLOMAN, M. 2003. An adaptive policy-based framework for network services management. *J. Network Syst. Manage. 11,* 3, 277–303.

MAMEI, M., MENEZES, R., TOLKSDORF, R., AND ZAMBONELLI, F. 2006. Case studies for self-organization in computer science. *Journal of Systems Architecture 52,* 8-9, 443–460.

MARANINCHI, F. AND RÉMOND, Y. 1998. Mode-automata: About modes and states for reactive systems. In *Proceedings of the 7th European Symposium on Programming (ESOP 1998)*, C. Hankin, Ed. Lecture Notes in Computer Science Series, vol. 1381. Springer, 185–199.

MARANINCHI, F. AND RÉMOND, Y. 2003. Mode-automata: a new domain-specific construct for the development of safe critical systems. *Sci. Comput. Program. 46,* 3, 219–254.

MARTÍ-OLIET, N., MESEGUER, J., AND VERDEJO, A. 2009. A rewriting semantics for maude strategies. *Electr. Notes Theor. Comput. Sci. 238,* 3, 227–247.

MCHUGH, J. 2007. Adaptive networks vision. ProCurve Networking, HP Innovation, available at `http://www.hp.com/md/pdfs/Adaptive_Networks_Vision_White_Paper.pdf`.

MCKINLEY, P. K., SADJADI, S. M., KASTEN, E. P., AND CHENG, B. H. C. 2004a. Composing adaptive software. *IEEE Computer 37,* 7, 56–64.

MCKINLEY, P. K., SADJADI, S. M., KASTEN, E. P., AND CHENG, B. H. C. 2004b. A taxonomy of compositional adaptation. Tech. Rep. MSU-CSE-04-17, Department of Computer Science, Michigan State University, East Lansing, Michigan. May.

MERELLI, E., PAOLETTI, N., AND TESEI, L. 2012. A multi-level model for self-adaptive systems. In *Proceedings of the 11th International Workshop on Foundations of Coordination Languages and Self Adaptation (FOCLASA 2012)*.

MESEGUER, J. 1992. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science 96,* 1, 73–155.

MESEGUER, J. AND TALCOTT, C. 2002. Semantic models for distributed object reflection. In *ECOOP 2002*, B. Magnusson, Ed. LNCS Series, vol. 2374. Springer, 1–36.

MIKIC-RAKIC, M. AND MEDVIDOVIC, N. 2006. A classification of disconnected operation techniques. In *Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA 2006)*. IEEE, 144–151.

MILNER, R. 1999. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press.

MONTESI, F., GUIDI, C., LUCCHI, R., AND ZAVATTARO, G. 2007. JOLIE: a Java orchestration language interpreter engine. *Electr. Notes Theor. Comput. Sci. 181*, 19–33.

MÜHL, G., WERNER, M., JAEGER, M., HERRMANN, K., AND PARZYJEGLA, H. 2002. On the Definitions of Self-Managing and Self-Organizing Systems. In *In T. Braun, G. Carle, and B. Stiller, editors, KiVS 2007 Workshop: Selbstorganisierende, Adaptive, Kontextsensitive verteilte Systeme (SAKS 2007)*. 291–301.

O'GRADY, R., CHRISTENSEN, A. L., PINCIROLI, C., AND DORIGO, M. 2010a. Robots autonomously self-assemble into dedicated morphologies to solve different tasks. In *AAMAS 2010*, W. van der Hoek, G. A. Kaminka, Y. Lespérance, M. Luck, and S. Sen, Eds. IFAAMAS, 1517–1518.

O'GRADY, R., GROSS, R., CHRISTENSEN, A. L., AND DORIGO, M. 2010b. Self-assembly strategies in a group of autonomous mobile robots. *Autonomous Robots 28,* 4, 439–455.

OREIZY, P., GORLICK, M. M., TAYLOR, R. N., HEIMBIGNER, D., JOHNSON, G., MEDVIDOVIC, N., QUILICI, A., ROSENBLUM, D. S., AND WOLF, A. L. 1999. An architecture-based approach to self-adaptive software. *Intelligent Systems and their Applications 14,* 3.

OUDSHOORN, M. J., FUAD, M. M., AND DEB, D. 2006. Towards autonomic computing: Injecting self-organizing and self-healing properties into java programs. In *Proceedings of the 2006 conference on New Trends in Software Methodologies, Tools and Techniques: Proceedings of the fifth SoMeT'06*. IOS Press, Amsterdam, The Netherlands, The Netherlands, 384–406.

PAVLOVIC, D. 2000. Towards semantics of self-adaptive software. In *Self-Adaptive Software, First International Workshop, IWSAS 2000, Oxford, UK, April 17-19, 2000, Revised Papers*, P. Robertson, H. E. Shrobe, and R. Laddaga, Eds. LNCS Series, vol. 1936. Springer, 65–74.

POPESCU, R., STAIKOPOULOS, A., BROGI, A., LIU, P., AND CLARKE, S. 2012. A formalized, taxonomy-driven approach to cross-layer application adaptation. *ACM Transactions on Autonomous and Adaptive Systems 7,* 1, 7:1–7:30.

POPOVICI, A., ALONSO, G., AND GROSS, T. R. 2003. Just-in-time aspects: efficient dynamic weaving for java. In *AOSD*. 100–109.

PUKALL, M., KÄSTNER, C., CAZZOLA, W., GÖTZ, S., GREBHAHN, A., AND REIMAR SCHRÖTER, G. S. 2012. Javadaptor: Flexible runtime updates of java applications. *Software: Practice and Experience*, n/a–n/a.

PUKALL, M., KÄSTNER, C., CAZZOLA, W., GÖTZ, S., GREBHAHN, A., SCHRÖTER, R., AND SAAKE, G. 2013. Javadaptor - flexible runtime updates of java applications. *Softw., Pract. Exper. 43,* 2, 153–185.

RAIBULET, C. 2008. Facets of adaptivity. In *Proceedings of the Second European Conference on Software Architecture (ECSA 2008)*, R. Morrison, D. Balasubramaniam, and K. E. Falkner, Eds. Lecture Notes in Computer Science Series, vol. 5292. Springer, 342–345.

RAMIREZ, A. J., CHENG, B. H. C., MCKINLEY, P. K., AND BECKMANN, B. E. 2010. Automatically generating adaptive logic to balance non-functional tradeoffs during reconfiguration. In *Proceedings of the 7th International Conference on Autonomic Computing, ICAC 2010, Reston, VA, USA, June 7-11, 2010*, M. Parashar, R. J. Figueiredo, and E. Kiciman, Eds. ACM, 225–234.

REDMOND, B. AND CAHILL, V. 2002. Supporting unanticipated dynamic adaptation of application behaviour. In *Proceedings of the 16th European Conference Object-Oriented Programming (ECOOP 2002)*, B. Magnusson, Ed. Lecture Notes in Computer Science Series, vol. 2374. Springer, 205–230.

SAGASTI, F. 1970. A conceptual and taxonomic framework for the analysis of adaptive behavior. *General Syst.* XV, 151160.

SALEHIE, M. AND TAHVILDARI, L. 2009. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems 4,* 2.

SALVANESCHI, G., GHEZZI, C., AND PRADELLA, M. An analysis of language-level support for self-adaptive software. *ACM Transactions on Autonomous and Adaptive Systems*.

SALVANESCHI, G., GHEZZI, C., AND PRADELLA, M. 2011. Context-oriented programming: A programming paradigm for autonomic systems. *CoRR abs/1105.0069*.

SANGIORGI, D. 1992. Expressing mobility in process algebras: First-order and higher-order paradigms. Ph.D. thesis, University of Edinburgh.

SCHAEFER, I. AND POETZSCH-HEFFTER, A. 2006. Using abstraction in modular verification of synchronous adaptive systems. In *Proceedings of the Workshop on "Trustworthy Software"*, S. Autexier, S. Merz, L. W. N. van der Torre, R. Wilhelm, and P. Wolper, Eds. OASICS Series, vol. 3. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany.

SCHNEIDER, K., SCHUELE, T., AND TRAPP, M. 2006. Verifying the adaptation behavior of embedded systems. In *Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems*. SEAMS '06. ACM, 16–22.

SECELEANU, T. AND GARLAN, D. 2006. Developing adaptive systems with synchronized architectures. *Journal of Systems and Software 79,* 11, 1514–1526.

TALCOTT, C. L. 2006. Coordination models based on a formal model of distributed object reflection. In *MTCoord 2005*, L. Brim and I. Linden, Eds. ENTCS Series, vol. 150(1). 143–157.

TALCOTT, C. L. 2007. Policy-based coordination in PAGODA: A case study. In *CoOrg 2006 & MTCoord 2006*, G. Boella, M. Dastani, A. Omicini, L. W. van der Torre, I. Cerna, and I. Linden, Eds. ENTCS Series, vol. 181. 97–112.

TARVAINEN, P. 2007. Adaptability evaluation of software architectures; a case study. In *Proceedings of the 31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*. IEEE Computer Society, 579–586.

VAN RENESSE, R., BIRMAN, K. P., HAYDEN, M., VAYSBURD, A., AND KARR, D. A. 1998. Building adaptive systems using ensemble. *Softw., Pract. Exper. 28,* 9, 963–979.

VAN RIEMSDIJK, M. B., MEYER, J.-J. C., AND DE BOER, F. S. 2006. Semantics of plan revision in intelligent agents. *Theor. Comput. Sci. 351,* 2, 240–257.

VINOGRADOV, R. AND SOKOLOV, V. 2010. On a class of high-level finite-state automata. *Automatic Control and Computer Sciences 44*, 398–406. 10.3103/S0146411610070059.

VIROLI, M., CASADEI, M., MONTAGNA, S., AND ZAMBONELLI, F. 2011. Spatial coordination of pervasive services through chemical-inspired tuple spaces. *ACM Transactions on Autonomous and Adaptive Systems 6,* 2, 14.

WANG, H., LV, H., AND FENG, G. 2009. A self-reflection model for autonomic computing systems based on π-calculus. In *NSS 2009*, Y. Xiang, J. Lopez, H. Wang, and W. Zhou, Eds. IEEE Computer Society, 310–315.

WEYNS, D., MALEK, S., AND ANDERSSON, J. 2010. FORMS: a formal reference model for self-adaptation. In *ICAC 2010*, R. Figueiredo and E. Kiciman, Eds. ACM, 205–214.

WEYNS, D., MALEK, S., AND ANDERSSON, J. 2012. FORMS: Unifying reference model for formal specification of distributed self-adaptive systems. *ACM Transactions on Autonomous and Adaptive Systems 7,* 1, 8.

WIRSING, M., HÖLZL, M., TRIBASTONE, M., AND ZAMBONELLI, F. ASCENS: Engineering autonomic service-component ensembles. To appear.

WIRTH, N. 1976. *Algorithms + Data Structures = Programs*. Prentice-Hall.

YANG, F., AOTANI, T., MASUHARA, H., NIELSON, F., AND NIELSON, H. R. 2011. Combining static analysis and runtime checking in security aspects for distributed tuple spaces. In *Proceedings of the 13th International Conference,Coordination Models and Languages (COORDINATION 2011)*, W. D. Meuter and G.-C. Roman, Eds. Lecture Notes in Computer Science Series, vol. 6721. Springer, 202–218.

ZADEH, L. A. 1963. On the definition of adaptivity. In *Proceedings of the IEEE*. Vol. 3. 469470.

ZHANG, J. AND CHENG, B. H. C. 2006a. Model-based development of dynamically adaptive software. In *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, L. J. Osterweil, H. D. Rombach, and M. L. Soffa, Eds. ACM, 371–380.

ZHANG, J. AND CHENG, B. H. C. 2006b. Using temporal logic to specify adaptive program semantics. *Journal of Systems and Software 79,* 10, 1361–1369.

ZHANG, J., GOLDSBY, H., AND CHENG, B. H. C. 2009. Modular verification of dynamically adaptive systems. In *Proceedings of the 8th International Conference on Aspect-Oriented Software Development (AOSD 2009)*, K. J. Sullivan, A. Moreira, C. Schwanninger, and J. Gray, Eds. ACM, 161–172.

ZHAO, Y., MA, D., LI, J., AND LI, Z. 2011a. Model checking of adaptive programs with mode-extended linear temporal logic. *IEEE International Workshop on Engineering of Autonomic and Autonomous Systems 0*, 40–48.

ZHAO, Y., MA, D., LI, J., AND LI, Z. 2011b. Model checking of adaptive programs with mode-extended linear temporal logic. In *8th IEEE International Conference and Workshops on Engineering of Autonomic and Autonomous Systems (EASe)*. IEEE Computer Society, 40–48.