A Conceptual Framework for Adaptation*

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

[1] Dipartimento di Informatica, Università di Pisa, Italy
[2] IMT Institute for Advanced Studies Lucca, Italy

Abstract. In this position paper we present a conceptual vision of adaptation, a key feature of autonomic systems. We put some stress on the role of control data and argue how some of the programming paradigms and models used for adaptive systems match with our conceptual framework.

Keywords: Adaptivity, autonomic systems, control data, MAPE-K control loop.

1 Introduction

Self-adaptive systems have been widely studied in several disciplines ranging from Biology to 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.

According to a widely accepted *black-box* or *behavioural* definition, a software system is called "self-adaptive" if it can modify its behaviour as a reaction to a change in its context of execution, understood in the widest possible way, including both the external environment and the internal state of the system itself. Typically the changes of behaviour are aimed at improving the degree of satisfaction of some either functional or non-functional requirements of the system, and self-adaptivity is considered a fundamental feature of *autonomic systems*, that can specialize to several other self-* properties (see e.g. [9]).

An interesting taxonomy is presented in [14], where the authors stress the highly interdisciplinary nature of the studies of such systems. Indeed, just restricting to the realm of Computer Science, active research on self-adaptive systems is carried out in Software Engineering, Artificial Intelligence, Control Theory, and Network and Distributed Computing, among others. However, as discussed in §3, only a few contributions address the foundational aspects of such systems, including their semantics and the use of formal methods for analysing them.

In this paper we propose an answer to very basic questions like **"when is a software system adaptive?"** or **"how can we identify the adaptation logic in an adaptive system?"**. We think that the limited effort placed in the investigation of the foundations of (self-)adaptive software systems might be due to the fact that it is not clear what are the characterizing features that distinguish

* Research supported by the European Integrated Project 257414 ASCENS.

© Springer-Verlag Berlin Heidelberg 2012

such systems from plain ("non-adaptive") ones. In fact, almost any software system can be considered self-adaptive, according to the black-box definition recalled above, since any system of a reasonable size can *modify its behaviour* (e.g., by executing different conditional branches) as a *reaction to a change in the context of execution* (like the change of variables or parameters).

These considerations show that the above behavioural definition of adaptivity is not useful in pinpointing adaptive systems, even if it allows to discard many systems that certainly are not. We should rather take a *white-box* perspective which allows us to inspect, to some extent, the internal structure of a system: we aim to have a clear *separation of concerns* to distinguish the cases where the changes of behaviour are part of the application logic from those where they realize the adaptation logic, calling adaptive only systems capable of the latter.

Self-adaptivity is often obtained by enriching the software that implements the standard application logic with a control loop that monitors the context of execution, determines the changes to be enforced, and enacts them. Systems featuring such an architectural pattern, often called MAPE-K [8,9,10], should definitely be considered as adaptive. But as argued in [4] there are other, simpler adaptive patterns, like the Internal Feedback Loop pattern, 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 adaptivity, 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 explicitely designed to express adaptation features. Archetypal examples are languages belonging to the paradigm of Context Oriented Programming, where the contexts of execution are first-class citizens [15], or to that of Dynamic Aspect Oriented Programming. Nevertheless, 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.

The goal of this position paper is to present a conceptual framework for adaptation, proposing a simple structural criterion to portray adaptivity (§2). We discuss how systems developed according to mainstream methodologies are shown to be adaptive according to our definition (§3), and explain how to understand adaptivity in many computational formalisms (§4). We sketch a first formalization of our concepts (§5). Finally, we discuss future developments of these ideas (§6).

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. Sometimes an adaptive behaviour of such a complex system may emerge from the interactions among its components, even if the components in isolation are not adaptive. However, we do not discuss this kind of adaptivity here: we focus instead on the adaptivity of simple components, for which we introduce the following conceptual framework.

According to a traditional paradigm, a program governing the behaviour of a component is made of *control* and *data*: these are two conceptual ingredients that in presence of sufficient resources (like computing power, memory or sensors) determine the behaviour of the component. In order to introduce adaptivity in this framework, we require to make explicit the fact that the behaviour of a component depends on some well identified *control data*. At this level of abstraction we are not concerned with the way the behaviour of the component is influenced by the control data, nor with the structure of such data.

Now, **we define *adaptation* as the run-time modification of the control data**. From this basic definition we derive several others. **A component is *adaptable* if it has a distinguished collection of control data that can be modified at run-time**. Thus if either the control data are not identified or they cannot be modified, then the system is not adaptable. Further, **a component is *adaptive* if it is adaptable and its control data are modified at run-time**, at least in some of its executions. And **a component is *self-adaptive* if it is able to modify its own control data at run-time**.

Given the intrinsic complexity of adaptive systems, this 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 adaptivity 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 "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, we may ask ourselves whether the execution of a simple branching statement, like **if** `tooHeavy` **then** `askForHelp` **else** `push` can be interpreted as a form of adaptation. The answer is: it depends.

Suppose that the statement is part of the software controlling a robot, and that the boolean variable `tooHeavy` is set according to the value returned by a sensor. If `tooHeavy` is considered as a *standard program variable*, then the change of behaviour caused by a change of its value is not considered "adaptation". If `tooHeavy` is instead considered as control data, then its change triggers an adaptation. Summing up, our question can be answered only after a clear identification of the control data.

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). We will come back to this methodological point along §3 and §4.

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 behaviour. Consequently, the nature of control data can vary considerably, ranging from simple configuration parameters to a complete representation of the program in execution that can be modified at run-time. This latter scenario 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 §4 how adaptivity, as defined above, can be obtained in systems implemented according to several computational formalisms. Before that, as a *proof of concept*, we discuss in the next section how several well accepted architectures of adaptive systems can be cast in our framework.

3 Architectures, Patterns and Reference Models for Adaptivity

Several contributions to the literature describe possible architectures or reference models for adaptive systems (or for *autonomic systems*, for which self-adaptivity is one of the main features). In this section we survey some of such proposals, stressing for each of them how a reasonable notion of control data can be identified. According to the MAPE-K architecture, a widely accepted reference model introduced in a seminal IBM paper [8], 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. Adaptation according to 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 either sensed by the monitor or modified by the execute phase of the control loop. Thus the control data represent the interface exposed by the managed component through which the control loop can operate, as shown in Fig. 1. The managed component is adaptable, and the system made of both the component and the control loop is self-adaptive.

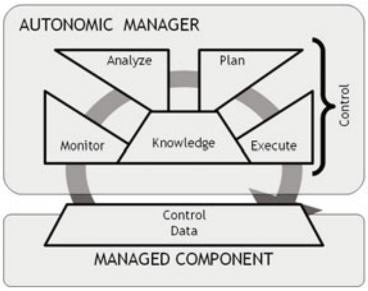

Fig. 1. Control data in MAPE-K

The construction can be iterated, as the control loop itself could be adaptable. As an example think of a component which follows a plan to perform some tasks. It 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. In this case, the planning component (that realizes the control loop of the base component)

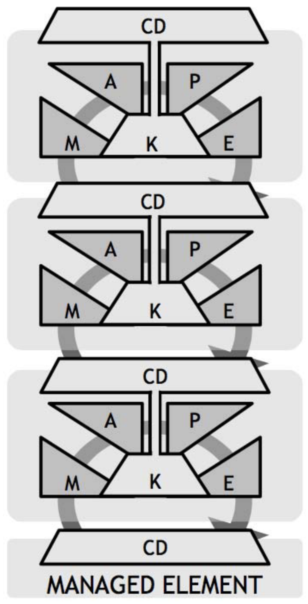

Fig. 2. Tower of adaptation

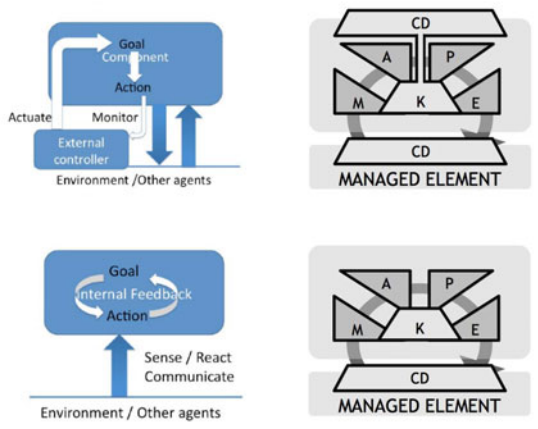

Fig. 3. External (top) and internal (bottom) patterns

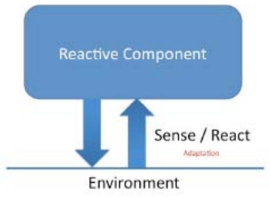

Fig. 4. Reactive pattern

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).

Another general reference model has been proposed in [1], 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, also 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 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, in [18] they introduce FORMS, a formal reference model that provides basic modeling primitives and relationships among them, suitable for the design of self-adaptive systems. Such primitives allow one to make explicit the presence of reflective (meta-level) subsystems, computations and models.

The goals of [1] are not dissimilar from ours, as they try to capture the essence of *self-adaptive* systems, identifying it in computational reflection; recall anyway that with our notion of control data we aimed at capturing the essence of the sole *adaptability*. We argue that in self-adaptive systems conforming to this model it should be relatively easy to identify the relevant control data. It is pretty clear that in reflective systems containing an explicit meta-model of the base-level system (like those conforming to the architecture-based solution proposed in [12]), such meta-model plays exactly the role of control data. Nevertheless, the FORMS modeling primitives can be instantiated and composed in a variety of ways (one for modeling MAPE-K and one for a specific application are discussed in [18]); in general in any such reflective system the control data could be identified at the boundaries between the meta-level and the base-level components.

In other frameworks for the design of adaptive systems (like [19]) the base-level system has a fixed collection of possible behaviours (or behavioural models), and adaptation consists of passing from one behaviour to another one, 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. The approach proposed in [19] 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`). 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* 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 and 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.

In the architectural approach of [2], 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/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.

More recently, a preliminary taxonomy of adaptive patterns has been proposed [4]. Two of these capture typical control loop patterns such as the *internal* and the *external* ones. Like MAPE-K, also these patterns can be cast easily in our framework (see Fig. 3): in the internal control loop pattern, the manager is a wrapper for the managed component and it is not adaptable, while in the external control loop pattern the manager is an adaptable component that is connected with the managed component. The third adaptive pattern describes *reactive* components (see Fig. 4). Such components are capable to modify 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.

Let us conclude by considering two of the few contributions that propose a formal semantics for adaptive systems. In [13] 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 coalgebric presentation potentially useful for further generalizations. At present it is not yet clear whether a notion of control data could fit in this abstract semantical framework or not: this is a topic of current investigation.

Finally, [3] proposes 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. A system is assumed to be deterministic, thus if it reacts non-deterministically to the input stream provided by the user, this 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 adaptivity is based on strong assumptions (e.g. systems are deterministic, all adaptive systems are interactive) that 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.

4 Adaptivity in Various Computational Paradigms

As observed in §2 and §3, 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 (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).

We outline some rules of thumb for the choice of control data within a few computational formalisms that are suited for implementing adaptive systems.

Context-Oriented Programming. Many programming languages have been promoted as suitable for programming adaptive systems [7]. A recent example is context-oriented programing which has been designed as a convenient paradigm for programming autonomic systems in general [15]. The main idea of this paradigm is that the execution of a program depends on the run-time 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. 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.

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*.

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 [15] and our ideas is that the former suggests the control data to reside within the manager, while we locate the control data in the interface of the managed component.

Declarative Programming. Logic programming 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-adaptivity, 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 machineries like term and graph rewriting. As many other programming paradigms, several rule-based approaches have been adapted or directly applied to adaptive systems (e.g. graph transformation [6]). 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 in such approaches are identified by the above mentioned separation of rules, or by the identification of the context-dependent conditions. Such identification is often not completely satisfactory and does not offer a neat and clear separation of concerns.

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. At the ground level, a rewrite theory $\mathcal{R}$ (e.g. software module) let us infer a computation step $\mathcal{R} \vdash t \rightarrow t'$ from a term (e.g. program state) t into t'. A universal theory $\mathcal{U}$ let 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*. This mechanism is efficiently supported by Maude [5] 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 [11] 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. 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 adaptation, 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 for modeling policy-based coordination [16] and for the design of PAGODA, a modular architecture for specifying autonomous systems [17].

Models of Concurrency. Languages and models emerged in the area of concurrency theory are natural candidates for the specification and analysis of autonomic systems. We inspect some (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 without doubts 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. Since the topology of the net is static, there is little margin 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 is part of 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 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 output parameters from some transition in the set of control places.

Dynamic nets allow for the creation of new subnets when certain transitions fire, so that the topology of the net can grow dynamically. Such "dynamic" transitions 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 within the hat of the interactive 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, the join calculus and other nominal calculi, including higher-order versions (e.g. the HO π-calculus) 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 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.

Paradigms with Reflective, Meta-level or Higher-Order Features. The same kind of adaptivity discussed for rewriting logic can be obtained in several other computational paradigms that, offering reflective, meta-level or higher-order features, allow one to represent programs as first-class citizens. In these cases adaptivity 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, besides rewriting logic, are process calculi with higher-order or meta-level aspects (e.g. HO π-calculus, MetaKlaim), higher-order variants of Petri nets and Graph Grammars, Logic Programming, and programming languages like LISP, Java, C#, Perl and several others. Systems implemented in these paradigms can realize adaptation within themselves (self-adaptivity), but in general the program under execution can be modified also by a different entity, like an autonomic control loop written in a different language, or in the same language but running in a separate thread.

5 A Formal Model for our Framework

We propose a simple formal model inspired by our conceptual framework. Our main purpose is to provide a proof-of-concept that validates the idea of developing formal models of adaptive systems where the key features of our approach (e.g. *control data*) are first-class citizens. The model we propose is deliberately simple and based on well-known computational artifacts, namely transition systems.

Overall Setting. We recall that a *labelled transition system* (LTS) is defined as a triple $L = (Q, A, \rightarrow)$ such that Q is the set of states, A is the alphabet of action labels and $\rightarrow \subseteq Q \times A \times Q$ is the transition relation. We write $q \xrightarrow{a} q'$ when $(q, a, q') \in \rightarrow$ and we say that the system can evolve from q to q' via action a. Sometimes, a distinguished initial state q_0 is also assumed.

The first ingredient is an LTS S that describes the behaviour of a software component. It is often the case that S is not running in isolation, but within a certain environment. The second ingredient is a LTS E that models the environment and that can constrain the computation of S, e.g. by forbidding certain actions and allowing others. We exploit the following composition operator over LTSs to define the behaviour of S within E as the LTS $S||E$.

Definition 1 (Composition). *Given two LTSs $L_1 = (Q_1, A_1, \rightarrow_1)$ and $L_2 = (Q_2, A_2, \rightarrow_2)$, we let $L_1||L_2$ denote the labelled transition system $(Q_1 \times Q_2, A_1 \cup A_2, \rightarrow)$, where $(q_1, q_2) \xrightarrow{a} (q_1', q_2')$ iff either of the following holds: $q_i \xrightarrow{a}_i q_i'$ for $i = 1, 2$ with $a \in A_1 \cap A_2$; $q_i \xrightarrow{a}_i q_i'$ and $q_j' = q_j$ for $\{i, j\} = \{1, 2\}$ with $a \in A_i \setminus A_j$.*

Note that in general it is not required that $A_1 = A_2$: the transitions are synchronised on common actions and are asynchronous otherwise.

Since adaptation is usually performed for the sake of improving a component's ability to perform some task or fulfill some goal, we provide here a very abstract but flexible notion of a component's objective in form of logical formulae. In particular, we let ψ be a formula (expressed in some suitable logic) characterizing the component's goal and we denote with the predicate $L \models \psi$ the property of the LTS L satisfying ψ. Note that it is not necessarily the case that $L \models \psi$ gives a yes/no result. For example, we may expect $L \models \psi$ to indicate how well L fits ψ, or the likelihood that L satisfies ψ. In the more general case, we can assume that $L \models \psi$ evaluates to a value in a suitable domain. We write $L \not\models \psi$ when L does not fit ψ, e.g. when the value returned is below a given threshold.

Adaptable vs non-adaptable Components. In a perfect but static world, one would engineer the software component S by ensuring that $S||E \models \psi$ and live happily afterwards (if such an S can be found). This is not realistic: the analyst has only a partial knowledge of E; S must be designed for running in different environments; the environment may change in an unpredictable manner by external causes while S is running; the goal ψ may be superseded by a more urgent goal ψ' to be accomplished. Roughly, we can expect frequent variations of E and possible, but less frequent, variations of ψ. The component is adaptable if it can cope with these changes in E and ψ by changes in its control data.

When S has no control data the component is not adaptable. The other extreme is when the whole S is the control data. Indeed an LTS can be represented and manipulated in several forms: as a list of transitions or as a transition matrix when it is finite; as a set of derivation rules when it is finitely specified.

Most appealing is the case when S is obtained as the combination of some statically fixed control FC and of some control data CD, i.e., $S = FC||CD$. Then, adaptavity is achieved by plugging-in a different control data CD' in reaction to a change in the environment from E to E' (with $S||E' \not\models \psi$ and $FC||CD'||E' \models \psi$), or to a change in the goal from ψ to ψ' (with $S||E \not\models \psi'$ and $FC||CD'||E \models \psi'$), or to a change in both.

We assume here that the managed component FC is determined statically such that it cannot be changed during execution and that each component may run under a unique manager CD at any time. However, adaptable components come equipped with a set of possible alternative managers $CD_1, ..., CD_k$ that can be determined statically or even derived dynamically during the computation.

Knowledge-Based Adaptation. Ideally, given FC, E and ψ it should be possible for the manager to select or construct the best suited control data CD_i (among the available choices) such that $FC||CD_i||E \models \psi$ and install it over FC. However, in real cases E may not be known entirely or may be so large that it is not convenient to represent it exactly. Therefore, we allow the manager to have a perfect knowledge of FC and of the goal ψ, but only a partial knowledge of E, that we denote by O and call the *observed environment*, or *context*.

The context O is derived by sensing the component's run-time environment. In general we cannot expect O and E to coincide: first, because the manager has limited sensing capabilities and second because the environment may be changed dynamically by other components after it has been sensed. Thus, O models the current perception of the environment from the viewpoint of the component.

The context O is expected to be updated frequently and to be used to adapt the component. This means that CD is chosen on the basis of FC, O and ψ, and that the manager can later discover that the real environment E differs from O in such a way that $FC||CD||E \not\models \psi$ even if $FC||CD||O \models \psi$. When this occurs, on the basis of the discovered discrepancies between E and O, a new context O' can be sensed to approximate E better than O, and O' can be used to determine some control data CD' in such a way that $FC||CD'||O' \models \psi$.

Self-adaptive Components. If the available control data strategies $CD_1, ...,$ CD_k are finitely many and statically fixed, then some precompilation can be applied that facilitates the adaptation to the changing environment, as explained below.

We assume that, given FC, ψ and any CD_i we can synthesize the weakest precondition ϕ_i on the environment such that $O \models \phi_i$ implies $FC||CD_i||O \models \psi$. Then, when the context changes from O to O', the manager can just look for some ϕ_j such that $O' \models \phi_j$ and then update the control data to CD_j.

Definition 2 (Self-adaptive Component). *A* self-adaptive component *is a tuple $\langle FC, \mathcal{CD}, \psi, \alpha_\psi \rangle$ where FC models the managed component; $\mathcal{CD}$ is a family of control data; ψ is the component's goal; and $\alpha_\psi : \mathcal{O} \times \mathcal{CD} \to \mathcal{CD}$ is a function that given a context $O \in \mathcal{O}$ and the current control data CD returns a control data CD' such that $FC||CD'||O \models \psi$.*

Enforcing the analogy of LTS based control, a possible formalization of the control manager of a self-adaptive component can be given as the composition of two LTSs: a fixed manager FM and the control data MCD defined as follows. The set of states of FM is $\mathcal{CD}$, and its transitions are labelled by context/goal pairs: for any CD, CD', O, ψ we have a transition $CD \xrightarrow{O,\psi} CD'$ iff $\alpha_\psi(O, CD) = CD'$. The LTS MCD has a single state and one looping transition labelled with the current context O and the current goal ψ. The composition $FM||MCD$ constrains the manager to ignore all transitions with labels different from O, ψ. The manager updates the control data of the managed component according to its current state. If CD' is the preferred strategy for O, ψ but CD is the current strategy, then the manager will move to CD' and then loop via $CD' \xrightarrow{O,\psi} CD'$.

Stacking Adaptive Components. Pushing our formal model further, by exploiting the control data of $\langle FC, \mathcal{CD}, \psi, \alpha \rangle$ we can add one more manager on top of the self-adaptive component, along the tower of adaptation (§3).

This second-level control manager can change the structure of MCD. For example, just by changing the label of its sole transition this (meta-)manager can model changes in the context, in the current goal, or in both.

However, one could argue that also other elements of the self-adaptive component could be considered as mutable. For example, one may want to change at run-time the adaptation strategy α_ψ that resolves the nondeterminism when there are several control data that can be successfully used to deal with the same context O, or even the set of available control data $\mathcal{CD}$, for example as the result of a learning mechanism. This can be formalized easily by exposing a larger portion of FM as control data.

Needless to say, also the above meta-manager can be designed as an adaptable component, formalizing its logic via a suitable LTS that exposes some control data to be managed by a upper level control manager, and so on.

6 Conclusion and Future Developments

We presented a conceptual framework for adaptation, where a central role is played by the explicit identification of the control data that govern the adaptive behavior of components. As a proof of concept we have discussed how systems conforming to well-accepted adaptive architectures, including IBM's MAPE-K and several adaptive patterns, fit into our framework. We have also considered several representative instances of our approach, focusing on foundational models of computation and programming paradigms, and we proposed a simple formalization of our concepts based on labelled transition systems.

We plan to exploit our conceptual framework by developing sound design principles for architectural styles and patterns in order to ensure correctness-by-design, and guidelines for the development of adaptive systems conforming to such patterns. For instance, we might think about imposing restrictions on the instances of our framework such as requiring an explicit separation of the component implementing the application logic from the component modifying the control data, in order to avoid self-adaptation within an atomic component and to guarantee separation of concerns, and an appropriate level of modularity.

We also plan to develop analysis and verification techniques for adaptive systems grounded on the central role of control data. Data- and control-flow analysis techniques could be used to separate, if possible, the adaptation logic from the application logic. This could also reveal the limits of our approach in situations where the adaptation and the application logics are too entangled.

Another current line of research aims at developing further the reflective, rule-based approach (§4). Starting from [11] we plan to use the Maude framework to develop prototype models of archetypal and newly emerging adaptive scenarios. The main idea is to exploit Maude's meta-programming facilities (based on logical reflection) and its formal toolset in order to specify, execute and analyze those prototype models. A very interesting road within this line is to equip Maude-programmed components with formal analysis capabilities like planning or model checking based on Maude-programmed tools.

Even if we focused the present work on adaptation issues of individual components, we also intend to develop a framework for adaptation of *ensembles*, i.e., massively parallel and distributed autonomic systems which act as a sort of swarm with emerging behavior. This could require to extend our *local* notion of control data to a *global* notion, where the control data of the individual components of an ensemble are treated as a whole, which will possibly require some mechanisms to amalgamate them for the manager, and to project them backwards to the components. Also, some mechanisms will be needed to coordinate the adaptation of individual components in order to obtain a meaningful adaptation of the whole system, in the spirit of the *overlapping adaptation* discussed in [19].

Last but not least, we intend to further investigate the connection of our work with the other approaches presented in the literature for adaptive, self-adaptive and autonomic systems: due to space limitation we have considered here just a few such instances.

References

1. Andersson, J., de Lemos, R., Malek, S., Weyns, D.: Reflecting on self-adaptive software systems. In: SEAMS 2009, pp. 38–47. IEEE Computer Society (2009)
2. Andrade, L.F., Fiadeiro, J.L.: An architectural approach to auto-adaptive systems. In: ICDCS Workshops 2002, pp. 439–444. IEEE Computer Society (2002)
3. Broy, M., Leuxner, C., Sitou, W., Spanfelner, B., Winter, S.: Formalizing the notion of adaptive system behavior. In: Shin, S.Y., Ossowski, S. (eds.) SAC 2009, pp. 1029–1033. ACM (2009)
4. Cabri, G., Puviani, M., Zambonelli, F.: Towards a taxonomy of adaptive agent-based collaboration patterns for autonomic service ensembles. In: Smari, W.W., Fox, G.C. (eds.) CTS 2011, pp. 508–515. IEEE Computer Society (2011)
5. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C. (eds.): All About Maude. LNCS, vol. 4350. Springer, Heidelberg (2007)
6. Ehrig, H., Ermel, C., Runge, O., Bucchiarone, A., Pelliccione, P.: Formal Analysis and Verification of Self-Healing Systems. In: Rosenblum, D.S., Taentzer, G. (eds.) FASE 2010. LNCS, vol. 6013, pp. 139–153. Springer, Heidelberg (2010)
7. Ghezzi, C., Pradella, M., Salvaneschi, G.: An evaluation of the adaptation capabilities in programming languages. In: Giese, H., Cheng, B.H. (eds.) SEAMS 2011, pp. 50–59. ACM (2011)
8. Horn, P.: Autonomic Computing: IBM's Perspective on the State of Information Technology (2001)
9. IBM Corporation: An Architectural Blueprint for Autonomic Computing (2006)
10. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer 36(1), 41–50 (2003)
11. Meseguer, J., Talcott, C.: Semantic Models for Distributed Object Reflection. In: Magnusson, B. (ed.) ECOOP 2002. LNCS, vol. 2374, pp. 1–36. Springer, Heidelberg (2002)
12. Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S., Wolf, A.L.: An architecture-based approach to self-adaptive software. Intelligent Systems and their Applications 14(3) (1999)
13. Pavlovic, D.: Towards Semantics of Self-Adaptive Software. In: Robertson, P., Shrobe, H.E., Laddaga, R. (eds.) IWSAS 2000. LNCS, vol. 1936, p. 50. Springer, Heidelberg (2001)
14. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Transactions on Autonomous and Adaptive Systems 4(2) (2009)
15. Salvaneschi, G., Ghezzi, C., Pradella, M.: Context-oriented programming: A programming paradigm for autonomic systems. CoRR abs/1105 0069 (2011)
16. Talcott, C.L.: Coordination models based on a formal model of distributed object reflection. In: Brim, L., Linden, I. (eds.) MTCoord 2005. ENTCS, vol. 150(1), pp. 143–157. Elsevier (2006)
17. Talcott, C.L.: Policy-based coordination in PAGODA: A case study. In: Boella, G., Dastani, M., Omicini, A., van der Torre, L.W., Cerna, I., Linden, I. (eds.) CoOrg 2006 & MTCoord 2006. ENTCS, vol. 181, pp. 97–112. Elsevier (2007)
18. Weyns, D., Malek, S., Andersson, J.: FORMS: a formal reference model for self-adaptation. In: Figueiredo, R., Kiciman, E. (eds.) ICAC 2010, pp. 205–214. ACM (2010)
19. Zhang, J., Cheng, B.H.C.: Model-based development of dynamically adaptive software. In: Osterweil, L.J., Rombach, H.D., Soffa, M.L. (eds.) ICSE 2006, pp. 371–380. ACM (2006)