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
[bruni,andrea,gadducci]@di.unipi.it
[2] IMT Institute for Advanced Studies Lucca, Italy
[alberto.lluch,andrea.vandin]@imtlucca.it

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. Still there is no general agreement about the foundational model of such systems.

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*, where such "context" has to be 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 so-called self-* properties (like self-configuration, self-optimization, self-protection and self-healing, as discussed for example in [6]).

An interesting taxonomy of the concepts related to self-adaptation is presented in [13], where the authors also stress the highly interdisciplinary nature of the studies of such systems. In fact, 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, only a few contributions address the foundational aspects of such systems, including their semantics or the use of formal methods for analysing them (see e.g. [12, 7]).

[⋆] Research supported by the European Integrated Project 257414 ASCENS

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 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. Indeed, any system of a reasonable size can *modify its behaviour* (e.g., executing different instructions, depending on conditional statements) as a *reaction to a change in the context of execution* (like the change of the value of a parameter, or the input of a data from the user).

These simple considerations show that the behavioural definition of adaptivity considered above is not useful in identifying which systems are adaptive, even if it allows to discard many systems that certainly are not. We aim to have a clear *separation of concerns* to distinguish situations where the modification of behaviour is part of the application logic from those where they realize the adaptation logic, calling adaptive only systems capable of the latter. We should rather consider a *white-box* perspective which allows us to inspect, at least to a certain extent, the internal structure of a system.

Self-adaptivity 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 enact them. Thus systems featuring such an architectural pattern, often called MAPE-K [5, 6, 8], should definitely be considered as adaptive. But as argued in [1] 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 these features. Paradigmatical examples are languages belonging to the paradigm of Context Oriented Programming, where the contexts of execution are first-class citizens [14], or to that of Dynamic Aspect Oriented Programming. Nevertheless, it is not the language that make 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 characterize adaptivity (§2). We discuss how systems developed according to mainstream methodologies should be easily shown to be adaptive according to our definition (§3), and explain how to understand adaptivity in many computational formalisms (§4). We also propose a first formalization of our concepts and apply it to an autonomic robot scenario (§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 precisely identified 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 assignment 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. We will come back to this point discussing several computational paradigms in §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.

Several well accepted architectures of adaptive systems can be cast in our framework, as discussed in §3. Just as an example, the IBM's reference model for adaptive systems, the MAPE-K control loop [5], can be considered as composed of a component implementing the application logic and exposing suitable control data, and of another component implementing the control loop and modifying the control data of the former.

The identification of the special role that control data play in a system could lead to better design principles and to novel analysis techniques for such systems, as discussed in §6.

3 Architectures and patterns for adaptivity

Starting with the seminal IBM paper [5] introducing the MAPE-K model, several contributionshave described possible architectures for adaptive systems (or for *autonomic systems*, for which self-adaptivity is one of the main features). According to the MAPE-K architecture, which is a widely accepted reference model, 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 components through which the control loop can operate, as shown in Fig. 1.

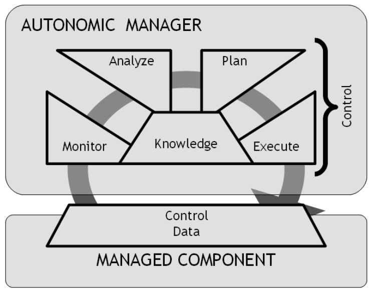

Fig. 1. Control data in MAPE-K.

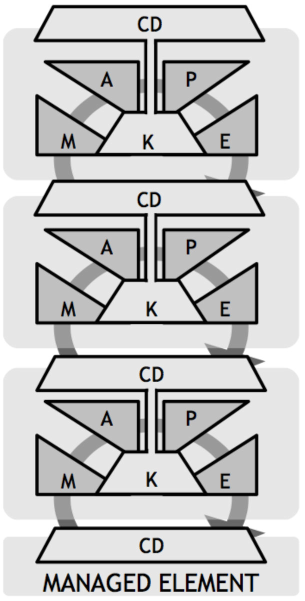

Fig. 2. Tower of adaptation.

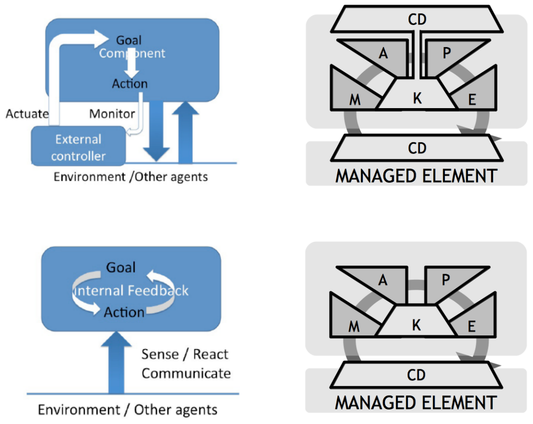

Fig. 3. External (top) and internal (bottom) patterns.

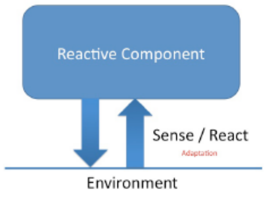

Fig. 4. Reactive pattern.

Clearly, by our definitions the managed component is adaptive, and the system made of both the component and the control loop is self-adaptive.

The construction can be iterated, as the control loop itself could be adaptive. As an example think of an adaptive component which follows a plan to perform some tasks. This component might have a manager which devises new plans according to changes in the context or in the component's goals. But this planning component might itself be adaptive, where some component 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 also the manager (the control loop) should expose in an interface its control data, which are conceptually part of its knowledge repository. In this way, the approach becomes compositional in a hierarchical way, which allows one to build towers of adaptive components (Fig. 2).

A richer architecture-based solution to adaptivity is proposed in [11]: at run-time, an architectural model is mantained that describes the running implementation. This model, made of components and connectors, can be modified by the control loop (adding or removing components or connectors, changing the topology and so on); an Evolution Manager ensures the consistency between the architectural model and the implementation, by changing the latter to reflect the changes to the former. In this case the architectural model plays the role of control data, as its modification triggers an adaptation of the implementation.

More recently, a preliminary taxonomy of adaptive patterns have been proposed [1]. 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 taxonomy of [1] includes a third adaptive pattern that describes *reactive* components (see Fig. 4). Such components are capable to modify their behavior in reaction to an external event, without any internal 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.

4 Adaptivity in various computational paradigms

As observed in §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 discuss a reasonable choice of control data for 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 [4]. A recent example is context-oriented programing which has been promoted as a suitable programming paradigm for autonomic systems in general [14]. Many languages have been extended to adopt the context oriented paradigm. We mention among others Lisp, Python, Ruby, Smalltalk, Scheme, Java, and Erlang. 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.

The notion of context varies from approach to approach but in general it refers 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 current active contexts, and variations can be used to propagate the invocation to the enclosing context scope.

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*.

The context stack acts then as the *control data* of our framework. With this view, the only difference between the approach proposed in [14] and our ideas is that the former suggests the control data to reside within the manager, while we promote the control data to reside in the managed component's interface.

More precisely, context-oriented programming can be used to instantiate our framework as follows: for each adaptable component (at any level of the adaptation tower) we must identify its possible contexts and implement the corresponding variations. The behavior of each such component must be under the scope of the component's context. The scope can enclose the whole behavior of the component or can be fine-grained at will, placing adaptation hooks where needed. In any case, the component should not be able to modify its context stack. Instead, the context stack should be made available for changes by the component's manager, which will be in charge of tuning the context stack after inspecting environmental information and the managed component's status.

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 goal 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 prefectly 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 [3]). Typical solutions include dividing the set of rules into those that correspond to ordinary computation 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 expressiveness feature of rewriting and logical frameworks like rewriting logic. In particular, the reflection mechanism of rewriting logic yields what is called the *tower of reflection.* 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 $\mathcal{U}$ lets us infer the computation at the meta-level, where theories and terms are meta-represented as terms: $\mathcal{U} \vdash (\overline{\mathcal{R}}, \overline{t}) \rightarrow (\overline{\mathcal{R}}, \overline{t'})$. The process can be repeated again and again as $\mathcal{U}$ itself is a rewrite theory. This mechanism is efficiently supported by Maude [2] and has given rise to many interesting meta-programming applications like analysis and transformation tools.

The tower of reflection immediately suggests a *tower of adaptation* as follows: At each level i, theories are composed by some immutable part $\mathcal{R}_i$ and some part subject to modification $\mathcal{CD}_i$, i.e. the control data. A natural choice for control data are subsets of rules. At the level 1, computation steps can modify both the term t and the control data $\mathcal{CD}_0$ of level 0: $\mathcal{U} + \mathcal{R}_1 + \mathcal{CD}_1 \vdash (\overline{R_0 + \mathcal{CD}_0}, \overline{t}) \rightarrow (\overline{R_0 + \mathcal{CD}'_0}, \overline{t'})$; and similarly for higher levels. It is easy to see that this tower of adaptation easily fits in our framework.

It is not an accident that logical reflection has been proposed as a suitable mean for modeling, specifying, verifying and executing adaptive systems. Some of the most representative examples are reported in [9] where the authors discuss various models of distributed (reflective) objects. One of the approaches described in [9] follows a *russian dolls* approach, essentially based on the use of nested components with logical reflection.

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 net allow for the creation of new subnets when certain transitions fire, so that the topology of the net can grow dynamically. In this case, besides the above considerations, 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 they 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 w.r.t. Petri nets, because processes will typically mix adaptation and computation.

The π-calculus, the join calculus and other nominal calculi, including higher-order versions (e.g. the $HO\pi$-calculus) can send and receive channels names, realizing some sort of reflexivity at the level of interaction: they have the ability to transmit transmission media. The situation is then analogous to that of dynamic nets, as new processes can be spawn in a way which is parametric w.r.t. the content of the received messages. If again we follow the distinction between adaptation channels names from ordinary channel names, then we need possibly 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 labelled transition systems. We apply it to a small scenario of an archetypal adaptive system.

Overall setting. We recall that a *labelled transition system* (LTS) is defined as a triple $L = (Q, A, \to)$ such that Q is the set of states, A is the alphabet of action labels and $\to \subseteq Q \times A \times Q$ is the transition relation. We write $q \xrightarrow{a} q'$ when $(q, a, q') \in \to$ 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.

We further recall that labelled transition systems are the semantic model of many system description languages such as various forms of automata, process calculi or programming languages and that there are various extensions for taking into account time, probability and other quantitative measures. Our presentation is however agnostic in this regard.

The first ingredient of our formal scenario 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. Thus, the second ingredient is another LTS E that we use to model 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* $L_1 = (Q_1, A_1, \to_1)$ *and* $L_2 = (Q_2, A_2, \to_2)$*, we let* $L_1||L_2$ *denote the labelled transition system* $(Q_1 \times Q_2, A_1 \cup A_2, \to)$*, 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.

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). But this is not realistic for several reasons: 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.

To one extreme, we can say that S has no control data, meaning the component is not adaptable.

The other extreme is when the whole S is *the* control data. In fact an LTS can be represented and manipulated in several forms: as list of transitions or as a transition matrix when it is finite; as a set of derivation rules when it is finitely specified. But this view would leave the component itself too loosely 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. 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 guessed 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 our tower of adaptation (§3).

Building on the above formalization, 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.

5.1 Modelling robot swarms

We consider the case study of a robot swarm whose goal is to collect some items on a unknown area. Scenarios of this kind are often called *fostering scenario* and are being proposed in various flavours in the literature of swarm computing in particular and adaptive systems in general (see e.g. [10]). Our own variant considers that the area to explore is represented as a (discrete) grid. The robots start in the origin of the grid (the *nest* of the swarm) and look around for items to be brought back to the nest.

The possible actions $A = \{n, s, w, e, l, u\}$ of the robots are to move towards north (n), south (s), west (w), east (e), to load an item (l) or to unload an item (u). The control FC of a robot has two states: explore (q_e) and return to the nest (q_r), with transitions: $q \xrightarrow{a} q$ for any $q \in \{q_e, q_r\}$ and any $a \in \{n, s, w, e\}$; $q_e \xrightarrow{l} q_r$ and $q_r \xrightarrow{u} q_e$.

The control data CD can be used to make the robot freely explore a certain area and make sure that if any item is found and loaded, then the robot goes back to the nest and then unloads the item.

For example, let $CD_1 = (Q_1, A, \rightarrow_1)$ be defined as $Q_1 = \{q_i \mid i \in [0, 10]\}$; $q_i \xrightarrow{n}_1 q_{i+1}$ and $q_{i+1} \xrightarrow{s}_1 q_i$ for $i \in [0, 9]$; $q_0 \xrightarrow{u}_1 q_0$ and $q_i \xrightarrow{l}_1 q_i$ for $i \in [1, 10]$. The control data CD_1 lets the robot explore the area only along a northbound path, up to ten cells far away from the nest. Similarly, let $CD_2 = (Q_2, A, \rightarrow_2)$ be defined as $Q_2 = \{q_i \mid i \in [0, 10]\}$; $q_i \xrightarrow{w}_2 q_{i+1}$ and $q_{i+1} \xrightarrow{e}_2 q_i$ for $i \in [0, 9]$; $q_0 \xrightarrow{u}_2 q_0$ and $q_i \xrightarrow{l}_2 q_i$ for $i \in [1, 10]$; The control data CD_2 lets the robot explore the area only along a westbound path, up to ten cells far away from the nest.

Let ψ be the formula expressing the ability to find an item to load within at most two cells away from the nest. Using the usual modal operators we can write the formula as $\psi = \psi_1 \vee \psi_2$ with $\psi_1 = \langle n, s, w, e \rangle \langle l \rangle \top$, and $\psi_2 = \langle n, s, w, e \rangle \psi_1$.

Then, the most general hypothesis on the context O w.r.t. CD_1 for $C || CD_1 || O \models \psi$ is that $O \models \phi_1$ with $\phi_1 = \langle n \rangle \langle l \rangle \top \vee \langle n \rangle \langle n \rangle \langle l \rangle \top$. Likewise, the most general hypothesis w.r.t. CD_2 is that $\phi_2 = \langle w \rangle \langle l \rangle \top \vee \langle w \rangle \langle w \rangle \langle l \rangle \top$.

Now, the context O observed by the robot may contain information gathered during previous movements, or perceived via sensors. It may include admissible directions for movements and even the (supposed) presence of items, which may have been observed (but not loaded) on the way back to the nest to unload some other item. The context O may not be faithful to E, because some obstacles may have been introduced or removed (e.g. other robots may have moved) and items may have been taken away in the meantime. Yet, O offers the basis to the manager on which to take the decision of using CD_1 or CD_2: if $O \models \phi_1$ and $O \not\models \phi_2$ then CD_1 is chosen; if $O \not\models \phi_1$ and $O \models \phi_2$ then CD_2 is chosen.

There are also two other cases. If $O \models \phi_i$ for $i = 1, 2$, then one may take a different interpretation for the logical $\vee$ and *count* the number of disjuncts being satisfied by O. For example, if two items can be found along the northbound path within three cells and only one along the westbound path within three cells, then CD_1 is to be preferred.

If no distinction can be made for O between ϕ_1 and ϕ_2, then a more accurate sensing of the environment can be tried, to get O' where a decision can be made.

When a robot swarm is considered, it could be useful to equip different robots with different sets of strategies for adaptation, so to force them to explore different, slightly overlapped, areas (to get a better covering and reduce the clash of robots), or to retrieve items at different distances (to avoid greedy behaviors at short distance which would leave far away items unobserved).

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 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. For example, data- and control-flow analysis techniques could be used to separate, if possible, the adaptation logic of a system (that modifies the control data) from the application logic (that just reads them).

Another current line of research aims at developing further the reflective, rule-based approach (§4). Starting from [9] 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 mechanism to amalgamate them for the manager, and to project them backwards to the components.

Last but not least, we intend to investigate the connection of our work with several 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 approaches.

References

1. 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)
2. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L.: All About Maude, LNCS, vol. 4350. Springer (2007)
3. Ehrig, H., Ermel, C., Runge, O., Bucchiarone, A., Pelliccione, P.: Formal analysis and verification of self-healing systems. In: Rosenblum, D., Taentzer, G. (eds.) FASE 2010, LNCS, vol. 6013, pp. 139–153. Springer (2010)
4. Ghezzi, C., Pradella, M., Salvaneschi, G.: An evaluation of the adaptation capabilities in programming languages. In: Proceeding of the 6th international symposium on Software engineering for adaptive and self-managing systems (SEAMS '11). ACM (2011)
5. Horn, P.: Autonomic Computing: IBM's perspective on the State of Information Technology (2001)
6. IBM Corporation: An Architectural Blueprint for Autonomic Computing (2006)
7. Karsai, G., Sztipanovits, J.: A model-based approach to self-adaptive software. Intelligent Systems and their Applications 14(3), 46–53 (1999)
8. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer 36(1), 41–50 (2003)
9. Meseguer, J., Talcott, C.: Semantic models for distributed object reflection. In: Magnusson, B. (ed.) ECOOP 2002, LNCS, vol. 2374, pp. 1–36. Springer (2002)
10. O'Grady, R., Christensen, A.L., Pinciroli, C., Dorigo, M.: Robots autonomously self-assemble into dedicated morphologies to solve different tasks. In: van der Hoek, W., Kaminka, G.A., Lespérance, Y., Luck, M., Sen, S. (eds.) AAMAS 2010. pp. 1517–1518. IFAAMAS (2010)
11. Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S., Wolf, A.L.: An architecture-based approach to self-adaptive software. Intelligent Systems and their Applications 14(3), 54–62 (1999)
12. Pavlovic, D.: Towards semantics of self-adaptive software. In: Robertson, P., Shrobe, H.E., Laddaga, R. (eds.) Proc. IWSAS'00. LNCS, vol. 1936, pp. 65–74. Springer (2000)
13. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Transactions on Autonomous and Adaptive Systems 4(2) (2009)
14. Salvaneschi, G., Ghezzi, C., Pradella, M.: Context-oriented programming: A programming paradigm for autonomic systems. CoRR abs/1105.0069 (2011)