Adaptable Transition Systems*

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

[1] Dipartimento di Informatica, University of Pisa, Italy
[bruni,andrea,gadducci]@di.unipi.it
[2] IMT Institute for Advanced Studies, Lucca, Italy
[alberto.lluch,andrea.vandin]@imtlucca.it

Abstract. We present an essential model of adaptable transition systems inspired by white-box approaches to adaptation and based on foundational models of component based systems. The key feature of adaptable transition systems are *control propositions*, imposing a clear separation between ordinary, functional behaviours and adaptive ones. We instantiate our approach on interface automata yielding *adaptable interface automata*, but it may be instantiated on other foundational models of component-based systems as well. We discuss how control propositions can be exploited in the specification and analysis of adaptive systems, focusing on various notions proposed in the literature, like *adaptability*, *control loops*, and *control synthesis*.

Keywords: Adaptation, autonomic systems, control data, interface automata

1 Introduction

Self-adaptive systems have been advocated as a convenient solution to the problem of mastering the complexity of modern software systems, networks and architectures. In particular, self-adaptivity is considered a fundamental feature of *autonomic systems*, that can specialise to several other self-* properties like self-configuration, self-optimisation, self-protection and self-healing. Despite some valuable efforts (see e.g. [16,11]), there is no general agreement on the notion of adaptivity, neither in general nor in software systems. There is as well no widely accepted foundational model for adaptivity. Using Zadeh's words [18]: *"it is very difficult -perhaps impossible- to find a way of characterizing in concrete terms the large variety of ways in which adaptive behavior can be realized"*. Zadeh's concerns were conceived in the field of Control Theory but are valid in Computer Science as well. Zadeh' skepticism for a concrete unifying definition of adaptivity is due to the attempt to subsume two aspects under the same definition: the *external* manifestations of adaptive systems (sometimes called *black-box* adaptation), and the *internal* mechanisms by which adaptation is achieved (sometimes called *white-box* adaptation).

* Research partially supported by the EU through the FP7-ICT Integrated Project 257414 ASCENS (Autonomic Service-Component Ensembles).

The limited effort placed so far in the investigation of the foundations of adaptive software systems might be due to the fact that it is not clear what are the characterising features that distinguish adaptive systems from those that are not so. For instance, very often a software system is considered "self-adaptive" if it "*modifies its own behavior in response to changes in its operating environment*" [14], when the software system realises that "*it is not accomplishing what the software is intended to do, or better functionality or performance is possible*" [15]. But, according to this definition, almost any software system can be considered self-adaptive, since any system of a reasonable complexity can *modify its behaviour* (e.g. following one of the different branches of a conditional statement) as a *reaction to a change in its context of execution* (e.g. values of variables or parameters).

Consider the automaton of Fig. 1, which models a server providing a task execution service. Each state has the format $s\{q\}[r]$ where s can be either `D` (the server is down) or `U` (it is up), and q, r are possibly empty sequences of `t` symbols representing, respectively, the lists of tasks scheduled for execution and the ones received but not scheduled yet. Transitions are labelled with `t?` (receive a task), `u!` (start-up the server), `s!` (schedule a task), `f!` (notify the conclusion of a task), and `d!` (shut-down the server). Annotations `?` and `!` denote *input* and *output* actions, respectively. Summing up, the server can receive tasks, start up, schedule tasks and notify their termination, and eventually shut down. Now, is the modelled server self-adaptive? One may argue that indeed it is, since the server schedules tasks only when it is up. Another argument can be that the server is self-adaptive since it starts up only when at least one task has to be processed, and shuts down only when no more tasks have to be processed. Or one could say that the server is not adaptive, because all transitions just implement its ordinary functional behaviour. Which is the right argument? How can we handle such diverse interpretations?

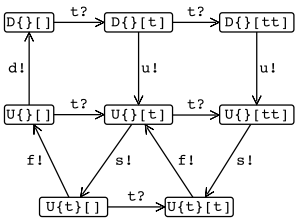

Fig. 1. Is it self-adaptive?

White-box adaptation. White-box perspectives on adaptation allow one to specify or inspect (part of) the internal structure of a system in order to offer a clear *separation of concerns* to distinguish changes of behaviour that are part of the application or functional logic from those which realise the adaptation logic.

In general, the behaviour of a component is governed by a program and according to the traditional, basic view, a program is made of *control* (i.e. algorithms) and *data*. The conceptual notion of adaptivity we proposed in [5] requires to identify *control data* which can be changed to *adapt* the component's behaviour. *Adaptation* is, hence, the run-time modification of such control data. Therefore, a component is *adaptable* if it has a distinguished collection of control data that can be modified at run-time, *adaptive* if it is adaptable and its control data are modified at run-time, at least in some of its executions, and *self-adaptive* if it modifies its own control data at run-time.

Several programming paradigms and reference models have been proposed for adaptive systems. A notable example is the Context Oriented Programming paradigm, where the contexts of execution and code variations are first-class citizens that can be used to structure the adaptation logic in a disciplined way [17]. Nevertheless, it is not the programming language what makes a program adaptive: 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 adaptation logic, that is the control data. Consequently, the nature of control data can vary considerably, including all possible ways of encapsulating behaviour: from simple configuration parameters to a complete representation of the program in execution that can be modified at run-time, as it is typical of computational models that support meta-programming or reflective features.

The subjectivity of adaptation is captured by the fact that the collection of control data of a component can be defined 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"). This means that white-box perspectives are as subjective as black-box ones. The fundamental difference lies in who is responsible of declaring which behaviours are part of the adaptation logic and which not: the observer (black-box) or the designer (white-box).

Consider again the system in Fig. 1 and the two possible interpretations of its adaptivity features. As elaborated in Sect. 3, in the first case control data is defined by the state of the server, while in the second case control data is defined by the two queues. If instead the system is not considered adaptive, then the control data is empty. This way the various interpretations are made concrete in our conceptual approach. We shall use this system as our running example.

It is worth to mention that the control data approach [5] is agnostic with respect to the form of interaction with the environment, the level of context-awareness, the use of reflection for self-awareness. It applies equally well to most of the existing approaches for designing adaptive systems and provides a satisfactory answer to the question "what is adaptation *conceptually*?". But "what is adaptation *formally*?" and "how can we reason about adaptation, *formally*?".

Contribution. This paper provides an answer to the questions we raised above. Building on our informal discussion, on a foundational model of component based systems (namely, *interface automata* [1,2], introduced in Sect. 2), and on previous formalisations of adaptive systems (discussed in Sect. 5) we distill in Sect. 3 a core model of adaptive systems called *adaptable interface automata* (AIAs). The key feature of AIAs are *control propositions* evaluated on states, the formal counterpart of control data. The choice of control propositions is arbitrary but it imposes a clear separation between ordinary, functional behaviours and adaptive ones. We then discuss in Sect. 4 how control propositions can be exploited in the specification and analysis of adaptive systems, focusing on various notions proposed in the literature, like *adaptability*, *feedback control loops*, and *control synthesis*. The approach based on control propositions can be applied to other computational models, yielding other instances of adaptable transition systems. The choice of interface automata is due to their simple and elegant theory.

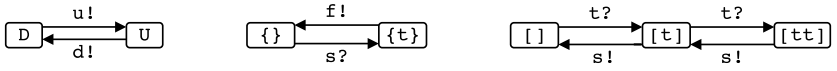

Fig. 2. Three interface automata: `Mac` (left), `Exe` (centre), and `Que` (right).

2 Background

Interface automata were introduced in [2] as a flexible framework for component-based design and verification. We recall here the main concepts from [1].

Definition 1 (interface automaton). *An interface automaton P is a tuple $\langle V, V^i, \mathcal{A}^I, \mathcal{A}^O, \mathcal{T} \rangle$, where V is a set of states; $V^i \subseteq V$ is the set of initial states, which contains at most one element (if V^i is empty then P is called* empty*); $\mathcal{A}^I$ and $\mathcal{A}^O$ are two disjoint sets of* input *and* output actions *(we denote by $\mathcal{A} = \mathcal{A}^I \cup \mathcal{A}^O$ the set of all actions); and $\mathcal{T} \subseteq V \times \mathcal{A} \times V$ is a deterministic set of* steps *(i.e. $(u, a, v) \in \mathcal{T}$, $(u, a, v') \in \mathcal{T}$ implies $v = v'$).*

Example 1. Figure 2 presents three interface automata modelling respectively a machine `Mac` (left), an execution queue `Exe` (centre), and a task queue `Que` (right). Intuitively, each automaton models one component of our running example (cf. Fig. 1). The format of the states is as in our running example. The initial states are not depicted on purpose, because we will consider several cases. Here we assume that they are `U`, {} and `[]`, respectively. The actions of the automata have been described in Sect. 1. The *interface* of each automaton is implicitly denoted by the action annotation: `?` for inputs and `!` for outputs.

Given $\mathcal{B} \subseteq \mathcal{A}$, we sometimes use $P_{|_\mathcal{B}}$ to denote the automaton obtained by restricting the set of steps to those whose action is in $\mathcal{B}$. Similarly, the set of actions in $\mathcal{B}$ labelling the outgoing transitions of a state u is denoted by $\mathcal{B}(u)$. A *computation* ρ of an interface automaton P is a finite or infinite sequence of consecutive *steps* (or *transitions*) $\{(u_i, a_i, u_{i+1})\}_{i<n}$ from $\mathcal{T}$ (thus n can be ω).

A partial composition operator is defined for automata: in order for two automata to be composable their interface must satisfy certain conditions.

Definition 2 (composability). *Let P and Q be two interface automata. Then, P and Q are* composable *if $\mathcal{A}^O_P \cap \mathcal{A}^O_Q = \emptyset$.*

Let $\mathit{shared}(P, Q) = \mathcal{A}_P \cap \mathcal{A}_Q$ and $\mathit{comm}(P, Q) = (\mathcal{A}^O_P \cap A^I_Q) \cup (\mathcal{A}^I_P \cap A^O_Q)$ be the set of *shared* and *communication* actions, respectively. Thus, two interface automata can be composed if they share input or communication actions only.

Two composable interface automata can be combined in a *product* as follows.

Definition 3 (product). *Let P and Q be two composable interface automata. Then the product $P \otimes Q$ is the interface automaton $\langle V, V^i, \mathcal{A}^I, \mathcal{A}^O, \mathcal{T} \rangle$ such that $V = V_P \times V_Q$; $V^i = V^i_P \times V^i_Q$; $\mathcal{A}^I = (\mathcal{A}^I_P \cup \mathcal{A}^I_Q) \setminus \mathit{comm}(P, Q)$; $\mathcal{A}^O = \mathcal{A}^O_P \cup \mathcal{A}^O_Q$; and*

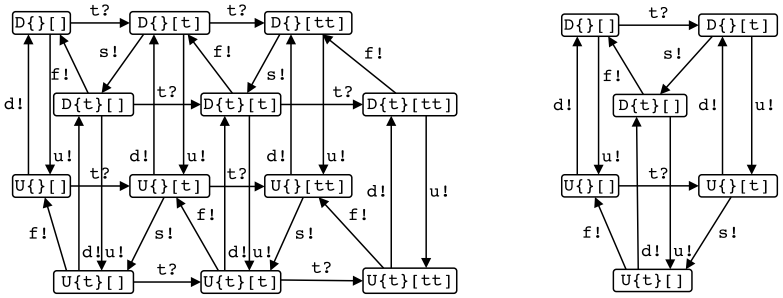

Fig. 3. The product `Mac` ⊗ `Exe` ⊗ `Que` (left) and the composition `Mac` | `Exe` | `Que` (right).

$\mathcal{T}$ is the union of $\{((v,u),a,(v',u)) \mid (v,a,v') \in \mathcal{T}_P \wedge a \notin \mathit{shared}(P,Q) \wedge u \in V_Q\}$ *(i.e. P steps),* $\{((v,u),a,(v,u')) \mid (u,a,u') \in \mathcal{T}_Q \wedge a \notin \mathit{shared}(P,Q) \wedge v \in V_P\}$ *(i.e. Q steps), and* $\{((v,u),a,(v',u')) \mid (v,a,v') \in \mathcal{T}_P \wedge (u,a,u') \in \mathcal{T}_Q \wedge a \in \mathit{shared}(P,Q)\}$ *(i.e. steps where P and Q synchronise over shared actions).*

In words, the product is a commutative and associative operation (up to isomorphism) that interleaves non-shared actions, while shared actions are synchronised in broadcast fashion, in such a way that shared input actions become inputs, communication actions become outputs.

Example 2. Consider the interface automata `Mac`, `Exe` and `Que` of Fig. 2. They are all pairwise composable and, moreover, the product of any two of them is composable with the remaining one. The result of applying the product of all three automata is depicted in Fig. 3 (left).

States in $P \otimes Q$ where a communication action is output by one automaton but cannot be accepted as input by the other are called *incompatible* or *illegal*.

Definition 4 (incompatible states). *Let P and Q be two composable interface automata. The set $\mathit{incompatible}(P,Q) \subseteq V_P \times V_Q$ of incompatible states of $P \otimes Q$ is defined as* $\{(u,v) \in V_P \times V_Q \mid \exists a \in \mathit{comm}(P,Q)\,.\,(a \in \mathcal{A}_P^O(u) \wedge a \notin \mathcal{A}_Q^I(v)) \vee (a \in \mathcal{A}_Q^O(v) \wedge a \notin \mathcal{A}_P^I(u))\}$.

Example 3. In our example, the product `Mac` ⊗ `Exe` ⊗ `Que` depicted in Fig. 3 (left) has several incompatible states, namely all those of the form "s{`t`}[`t`]" or "s{`t`}[`tt`]". Indeed, in those states, `Que` is willing to perform the output action `s!` but `Exe` is not able to perform the dual input action `s?`.

The presence of incompatible states does not forbid to compose interface automata. In an open system, compatibility can be ensured by a third automata called the *environment* which may e.g. represent the context of execution or an adaptation manager. Technically, an environment for an automaton R is a non-empty automaton E which is composable with R, synchronises with all output actions of R (i.e. $\mathcal{A}_E^I = \mathcal{A}_R^O$) and whose product with R does not have incompatible states. Interesting is the case when R is $P \otimes Q$ and E is a *compatible* environment, i.e. when the set $\mathit{incompatible}(P,Q) \times V_E$ is not reachable in $R \otimes E$.

Compatibility of two (composable, non-empty) automata is then expressed as the existence of a compatible environment for them. This also leads to the concept of *compatible* (or *usable*) states $cmp(P \otimes Q)$ in the product of two composable interface automata P and Q, i.e. those for which an environment E exists that makes the set of incompatible states $incompatible(P, Q)$ unreachable in $P \otimes Q \otimes E$.

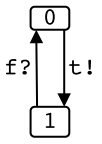

Fig. 4. An environment.

Example 4. Consider again the interface automata `Mac`, `Exe` and `Que` of Fig. 2. Automata `Mac` and `Exe` are trivially compatible, and so are `Mac` and `Que`. `Exe` and `Que` are compatible as well, despite of the incompatible states `{t}[t]` and `{t}[tt]` in their product `Exe` $\otimes$ `Que`. Indeed an environment that does not issue a second task execution requests `t!` without first waiting for a termination notification (like the one in Fig. 4) can avoid reaching the incompatible states.

We are finally ready to define the composition of interface automata.

Definition 5 (composition). *Let P and Q be two composable interface automata. The composition $P \mid Q$ is an interface automaton $\langle V, V^i, \mathcal{A}^I_{P\otimes Q}, \mathcal{A}^O_{P\otimes Q}, \mathcal{T}\rangle$ such that $V = cmp(P \otimes Q)$; $V^i = V^i_{P\otimes Q} \cap V$; and $\mathcal{T} = \mathcal{T}_{P\otimes Q} \cap (V \times \mathcal{A} \times V)$.*

Example 5. Consider the product `Mac` $\otimes$ `Exe` $\otimes$ `Que` depicted in Fig. 3 (left). All states of the form s`{t}[t]` and s`{t}[tt]` are incompatible and states `D{}[tt]` and `U{}[tt]` are not compatible, since no environment can prevent them to enter the incompatible states. The remaining states are all compatible. The composition `Mac` | `Exe` | `Que` is the interface automaton depicted in Fig. 3 (right).

3 Adaptable Interface Automata

Adaptable interface automata extend interface automata with atomic propositions (state observations) a subset of which is called *control propositions* and play the role of the control data of [5].

Definition 6 (adaptable interface automata). *An* adaptable interface automaton *(*AIA*) is a tuple $\langle P, \Phi, l, \Phi^c\rangle$ such that $P = \langle V, V^i, \mathcal{A}^I, \mathcal{A}^O, \mathcal{T}\rangle$ is an interface automaton; Φ is a set of atomic propositions, $l : V \to 2^\Phi$ is a labelling function mapping states to sets of propositions; and $\Phi^c \subseteq \Phi$ is a distinguished subset of* control propositions*.*

Abusing the notation we sometimes call P an AIA with underlying interface automaton P, whenever this introduces no ambiguity. A transition $(u, a, u') \in T$ is called an *adaptation* if it changes the control data, i.e. if there exists a proposition $\phi \in \Phi^c$ such that either $\phi \in l(u)$ and $\phi \notin l(u')$, or vice versa. Otherwise, it is called a *basic* transition. An action $a \in A$ is called a *control action* if it labels at least one adaptation. The set of all control actions of an AIA P is denoted by $\mathcal{A}^C_P$.

Example 6. Recall the example introduced in Sect. 1. We raised the question whether the interface automaton S of Fig. 1 is (self-)adaptive or not. Two arguments were given. The first argument was "*the server schedules tasks only when it is up*". That is, we identify two different behaviours of the server (when it is up or down, respectively), interpreting a change of behaviour as an adaptation. We can capture this interpretation by introducing a control proposition that records the state of the server. More precisely, we define the AIA Switch(S) in the following manner. The underlying interface automaton is S; the only (control) proposition is *up*, and the labelling function maps states of the form U{...}[...] into {*up*} and those of the form D{...}[...] into ∅. The control actions are then u and d. The second argument was "*the system starts the server up only when there is at least one task to schedule, and shuts it down only when no task has to be processed*". In this case the change of behaviour (adaptation) is triggered either by the arrival of a task in the waiting queue, or by the removal of the last task scheduled for execution. Therefore we can define the control data as the state of both queues. That is, one can define an AIA Scheduler(S) having as underlying interface automaton the one of Fig. 1, as control propositions all those of the form *queues_status_q_r* (with $q \in \{\_, \mathtt{t}\}$, and $r \in \{\_, \mathtt{t}, \mathtt{tt}\}$), and a labelling function that maps states of the form $s\{q\}[r]$ to the set $\{queues\_status\_q\_r\}$. In this case the control actions are s, f and t.

Computations. The computations of an AIA (i.e. those of the underlying interface automata) can be classified according to the presence of adaptation transitions. For example, a computation is *basic* if it contains no adaptive step, and it is *adaptive* otherwise. We will also use the concepts of *basic computation* starting at a state u and of *adaptation phase*, i.e. a maximal computation made of adaptive steps only.

Coherent control. It is worth to remark that what distinguishes adaptive computations and adaptation phases are not the actions, because control actions may also label transitions that are not adaptations. However, very often an AIA has *coherent control*, meaning that the choice of control propositions is coherent with the induced set of control actions, in the sense that all the transitions labelled with control actions are adaptations.

Composition. The properties of composability and compatibility for AIA, as well as product and composition operators, are lifted from interface automata.

Definition 7 (composition). *Let P and Q be two* AIA*s whose underlying interface automata P', Q' are composable. The composition $P \mid Q$ is the* AIA $\langle P' \mid Q', \Phi, l, \Phi^c \rangle$ *such that the underlying interface automaton is the composition of P' and Q'; $\Phi = \Phi_P \uplus \Phi_Q$ (i.e. the set of atomic propositions is the disjoint union of the atomic propositions of P and Q); $\Phi^c = \Phi^c_P \uplus \Phi^c_Q$; and l is such that $l((u,v)) = l_P(u) \cup l_Q(v)$ for all $(u,v) \in V$ (i.e. a proposition holds in a composed state if it holds in its original local state).*

Since the control propositions of the composed system are the disjoint union of those of the components, one easily derives that control coherence is preserved by composition, and that the set of control actions of the product is obtained as the union of those of the components.

4 Exploiting Control Data

We explain here how the distinguishing features of AIA (i.e. control propositions and actions) can be exploited in the design and analysis of self-adaptive systems. For the sake of simplicity we will focus on AIA with coherent control, as it is the case of all of our examples. Thus, all the various definitions/operators that we are going to define on AIA may rely on the manipulation of control actions only.

4.1 Design

Well-formed interfaces. The relationship between the set of control actions $\mathcal{A}_P^C$ and the alphabets $\mathcal{A}_P^I$ and $\mathcal{A}_P^O$ is arbitrary in general, but it could satisfy some pretty obvious constraints for specific classes of systems.

Definition 8 (adaptable, controllable and self-adaptive ATSs). *Let P be an* AIA*. We say that P is* adaptable *if $\mathcal{A}_P^C \neq \emptyset$;* controllable *if $\mathcal{A}_P^C \cap \mathcal{A}_P^I \neq \emptyset$;* self-adaptive *if $\mathcal{A}_P^C \cap \mathcal{A}_P^O \neq \emptyset$.*

Intuitively, an AIA is *adaptable* if it has at least one control action, which means that at least one transition is an adaptation. An adaptable AIA is *controllable* if control actions include some input actions, or *self-adaptive* if control actions include some output actions (which are under control of the AIA).

From these notions we can derive others. For instance, we can say that an adaptable AIA is *fully self-adaptive* if $\mathcal{A}_P^C \cap \mathcal{A}_P^I = \emptyset$ (the AIA has full control over adaptations). Note that hybrid situations are possible as well, when control actions include both input actions (i.e. actions in $\mathcal{A}_P^I$) and output actions (i.e. actions in $\mathcal{A}_P^O$). In this case we have that P is both *self-adaptive* and *controllable*.

Example 7. Consider the AIA `Scheduler(S)` and `Switch(S)` described in Example 6, whose underlying automaton (`S`) is depicted in Fig. 1. `Switch(S)` is fully self-adaptive and not controllable, since its control actions do not include input actions, and therefore the environment cannot force the execution of control actions directly. On the other hand, `Scheduler(S)` is self-adaptive and controllable, since some of its control actions are outputs and some are inputs.

Consider instead the interface automaton `A` in the left of Fig. 5, which is very much like the automaton $\texttt{Mac} \otimes \texttt{Exe} \otimes \texttt{Que}$ of Fig. 3, except that all actions but `f` have been turned into input actions and states of the form $s\{\texttt{t}\}[\texttt{tt}]$ have been removed. The automaton can also be seen as the composition of the two automata on the right of Fig. 5. And let us call `Scheduler(A)` and `Switch(A)` the AIA obtained by applying the control data criteria of `Scheduler(S)` and `Switch(S)`, respectively. Both `Scheduler(A)` and `Switch(A)` are adaptable and controllable, but only `Scheduler(A)` is self-adaptive, since it has at least one control output action (i.e. `f!`).

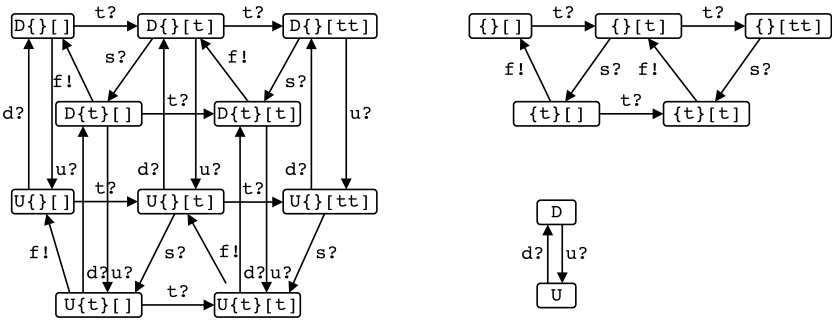

Fig. 5. An adaptable server (left) and its components (right).

Composition. As discussed in Sect. 3, the composition operation of interface automata can be extended seamlessly to AIA. Composition can be used, for example, to combine an adaptable basic component B and an adaptation manager M in a way that reflects a specific adaptation logic. In this case, natural well-formedness constraints can be expressed as suitable relations among sets of actions. For example, we can define when a component M controls another component B as follows.

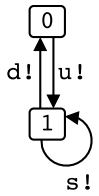

Fig. 6. A controller.

Definition 9 (controlled composition). *Let B and M be two composable* AIA. *We say that M* controls *B in $B \mid M$ if $\mathcal{A}_B^C \cap \mathcal{A}_M^O \neq \emptyset$. In addition, we say that M controls* completely *B in $B \mid M$ if $\mathcal{A}_B^C \subseteq \mathcal{A}_M^O$.*

This definition can be used, for instance, to allow or to forbid mutual control. For example, if a manager M is itself at least partly controllable (i.e. $\mathcal{A}_M^C \cap \mathcal{A}_M^I \neq \emptyset$), a natural requirement to avoid mutual control would be that the managed component B and M are such the $\mathcal{A}_B^O \cap \mathcal{A}_M^C = \emptyset$, i.e. that B cannot control M.

Example 8. Consider the adaptable server depicted on the left of Fig. 5 as the basic component whose control actions are `d`, `u` and `s`. Consider further the controller of Fig. 6 as the manager, which controls completely the basic component. A superficial look at the server and the controller may lead to think that their composition yields the adaptive server of Fig. 1, yet this not the case. Indeed, the underlying interface automata are not compatible due to the existence of (unavoidable) incompatible states.

Control loops and action classification. The distinction between input, output and control actions is suitable to model some basic interactions and well-formedness criteria as we explained above. More sophisticated cases such as control loops are better modelled if further classes of actions are distinguished.

As a paradigmatic example, let us consider the control loop of the MAPE-K reference model [9], illustrated in Fig. 7. This reference model is the most influential one for autonomic and adaptive systems. The name MAPE-K is due to the main activities of autonomic manager components (Monitor, Analyse, Plan, Execute) and the fact that all such activities operate and exploit the same Knowledge base.

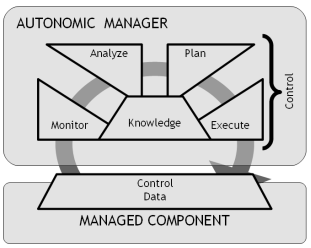

Fig. 7. MAPE-K loop.

According to this 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. The managed component is considered to be an adaptable component, and the system made of both the component and the manager implementing the control loop is considered as a self-adaptive component.

AIA can be composed so to adhere to the MAPE-K reference model as schematised in Fig. 8. First, the autonomic manager component M and the managed component B have their *functional* input and output actions, respectively $I \subseteq \mathcal{A}^I_M$, $O \subseteq \mathcal{A}^O_M$, $I' \subseteq \mathcal{A}^I_B$, $O' \subseteq \mathcal{A}^O_B$ such that no dual action is shared (i.e. $comm(B, M) \cap (I \cup I') = \emptyset$) but inputs may be shared (i.e. possibly $I \cap I' \neq \emptyset$). The autonomic manager is controllable and has hence a distinguished set of control actions $C = \mathcal{A}^C_B$. The dual of such control actions, i.e. the output actions of M that synchronise with the input control actions B can be regarded as *effectors* $F \subseteq \mathcal{A}^O_M$, i.e. output actions used to trigger adaptation. In addition, M will also have *sensor* input actions $S \subseteq \mathcal{A}^I_M$ to sense the status of B, notified via *emit* output actions $E \subseteq \mathcal{A}^O_M$. Clearly, the introduced sets partition inputs and outputs, i.e. $I \uplus S = \mathcal{A}^I_M$, $O \uplus F = \mathcal{A}^O_M$, $E \uplus I' = \mathcal{A}^I_B$ and $O' \uplus C = \mathcal{A}^O_M$.

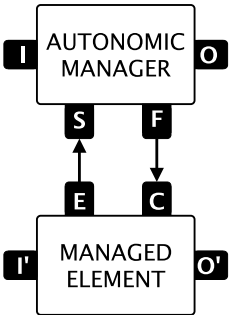

Fig. 8. MAPE-K actions.

4.2 Analysis and verification

Property classes. By the very nature of adaptive systems, properties that one is interested to verify on them can be classified according to the kind of computations that are concerned with, so that the usual verification (e.g. model checking problem) $P \models \psi$ (i.e. *"does the* AIA P *satisfy property* ψ*?"*) is instantiated in some of the computations of P depending of the class of ψ.

For example, some authors (e.g. [20,19,10]) distinguish the following three kinds of properties. *Local* properties are "*properties of one [behavioral] mode*", i.e. properties that must be satisfied by basic computations only. *Adaptation* properties are to be "*satisfied on interval states when adapting from one behavioral mode to another*", i.e. properties of adaptation phases. *Global* properties "*regard program behavior and adaptations as a whole. They should be satisfied by*

the adaptive program throughout its execution, regardless of the adaptations.", i.e. properties about the overall behaviour of the system.

To these we add the class of *adaptability* properties, i.e. properties that may fail for local (i.e. basic) computations, and that need the adapting capability of the system to be satisfied.

Definition 10 (adaptability property). *Let P be an* AIA*. A property ψ is an* adaptability *property for P if $P \models \psi$ and $P_{|_{\mathcal{A}_P \setminus \mathcal{A}_P^C}} \not\models \psi$.*

Example 9. Consider the adaptive server of Fig. 1 and the AIA `Scheduler(S)` and `Switch(S)`, with initial state `U{}[]`. Consider further the property "*whenever a task is received, the server can finish it*". This is an adaptability property for `Scheduler(S)` but not for `Switch(S)`. The main reason is that in order to finish a task it first has to be received (`t`) and scheduled (`s`), which is part of the adaptation logic in `Scheduler(S)` but not in `Switch(S)`. In the latter, indeed, the basic computations starting from state `U{}[]` are able to satisfy the property.

Weak and strong adaptability. AIA are also amenable for the analysis of the computations of interface automata in terms of adaptability. For instance, the concepts of *weak* and *strong* adaptability from [13] can be very easily rephrased in our setting. According to [13] a system is *weakly adaptable* if "*for all paths, it always holds that as soon as adaptation starts, there exists* at least one *path for which the system eventually ends the adaptation phase*", while a system is *strongly adaptable* if "*for all paths, it always holds that as soon as adaptation starts,* all *paths eventually end the adaptation phase*".

Strong and weak adaptability can also be characterised by formulae in some temporal logic [13], ACTL [7] in our setting.

Definition 11 (weak and strong adaptability). *Let P be an* AIA*. We say that P is* weakly adaptable *if $P \models$ **AG EF EX**$\{\mathcal{A}_P \setminus \mathcal{A}_P^C\}true$, and* strongly adaptable *if $P \models$ **AG AF** (**EX**$\{\mathcal{A}_P\}true \wedge$ **AX**$\{\mathcal{A}_P \setminus \mathcal{A}_P^C\}true$).*

The formula characterising weak adaptability states that along all paths (**A**) it always (**G**) holds that there is a path (**E**) where eventually (**F**) a state will be reached where a basic step can be executed (**EX**$\{\mathcal{A}_P \setminus \mathcal{A}_P^C\}true$). Similarly, the formula characterising strong adaptability states that along all paths (**A**) it always (**G**) holds that along all paths (**A**) eventually (**F**) a state will be reached where at least one step can be fired (**EX**$\{\mathcal{A}_P\}true$) and all fireable actions are basic steps (**AX**$\{\mathcal{A}_P \setminus \mathcal{A}_P^C\}true$). Apart from its conciseness, such characterisations enables the use of model checking techniques to verify them.

Example 10. The AIA `Switch(S)` (cf. Fig. 1) is strongly adaptable, since it does not have any infinite adaptation phase. Indeed every control action (`u` or `d`) leads to a state where only basic actions (`t`, `f` or `s`) can be fired. On the other hand, `Scheduler(S)` is weakly adaptable due to the presence of loops made of adaptive transitions only (namely, `t`, `s` and `f`), which introduce an infinite adaptation

phase. Consider now the AIA `Scheduler(A)` and `Switch(A)` (cf. Fig. 5). Both are weakly adaptable due to the loops made of adaptive transitions only;: e.g. in `Switch(A)` there are cyclic behaviours made of the control actions `u` and `d`.

4.3 Reverse engineering and control synthesis

Control data can also guide reverse engineering activities. For instance, is it possible to decompose an AIA S into a basic adaptable component B and a suitable controller M? We answer in the positive, by presenting first a trivial solution and then a more sophisticated one based on control synthesis.

Basic decomposition. In order to present the basic decomposition we need some definitions. Let $P^{\perp_{\mathcal{B}}}$ denote the operation that given an automaton P results in an automaton $P^{\perp_{\mathcal{B}}}$ which is like P but where actions in $\mathcal{B} \subseteq \mathcal{A}$ have been complemented (inputs become outputs and vice versa). Formally, $P^{\perp_{\mathcal{B}}} = \langle V, V^i, ((\mathcal{A}^I \setminus \mathcal{B}) \cup (\mathcal{A}^O \cap \mathcal{B})), ((\mathcal{A}^O \setminus \mathcal{B}) \cup (\mathcal{A}^I \cap \mathcal{B})), \mathcal{T} \rangle$. This operation can be trivially lifted to AIA by preserving the set of control actions.

It is easy to see that interface automata have the following property. If P is an interface automaton and O_1, O_2 are sets of actions that partition $\mathcal{A}^O_P$ (i.e. $\mathcal{A}^O_P = O_1 \uplus O_2$), then P is isomorphic to $P^{\perp_{O_1}} \mid P^{\perp_{O_2}}$. This property can be exploited to decompose an AIA P as $M \mid B$ by choosing $M = P^{\perp_{\mathcal{A}^O_P \setminus \mathcal{A}^C_P}}$ and $B = P^{\perp_{\mathcal{A}^O_P \cap \mathcal{A}^C_P}}$. Intuitively, the manager and the base component are identical to the original system and only differ in their interface. All output control actions are governed by the manager M and become inputs in the base component B. Outputs that are not control actions become inputs in the manager. This decomposition has some interesting properties: B is fully controllable and, if P is fully self-adaptive, then M completely controls B.

Example 11. Consider the server `Scheduler(S)` (cf. Fig. 1). The basic decomposition provides the manager with underlying automata depicted in Fig. 9 (left) and the basic component depicted in Fig. 9 (right). Vice versa, if the server `Switch(S)` (cf. Fig. 1) is considered, then the basic decomposition provides the manager with underlying automata depicted in Fig. 9 (right) and the basic component depicted in Fig. 9 (left).

Decomposition as control synthesis. In the basic decomposition both M and B are isomorphic (and hence of equal size) to the original AIA S, modulo the complementation of some actions. It is however possible to apply heuristics in order to obtain smaller non-trivial managers and base components. One possibility is to reduce the set of actions that M needs to observe (its input actions). Intuitively, one can make the choice of ignoring some input actions and collapse the corresponding transitions. Of course, the resulting manager M must be checked for the absence of non-determinism (possibly introduced by the identification of

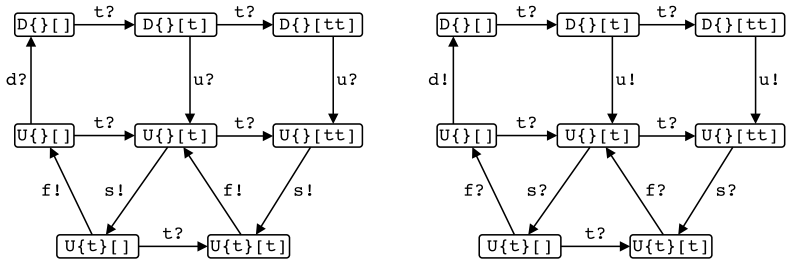

Fig. 9. A basic decomposition.

states) but will be a smaller manager candidate. Once a candidate M is chosen we can resort to solutions to the control synthesis problem.

We recall that the synthesis of controllers for interface automata [4] is the problem of solving the equation $P \mid Y \preceq Q$, for a given system Q and component P, i.e. finding a component Y such that, when composed with P, results in a system which *refines* Q. An interface automaton R *refines* an interface automaton S if (i) $\mathcal{A}_R^I \subseteq \mathcal{A}_S^I$, (ii) $\mathcal{A}_R^O \subseteq \mathcal{A}_S^O$, and (iii) there is an *alternating simulation* relation ϱ from R to S, and two states $u \in V_R^i$, $v \in V_S^i$ such that $(u, v) \in \varrho$ [1]. An *alternating simulation* relation ϱ from an interface automaton R to an interface automaton S is a relation $\varrho \subseteq V_R \times V_S$ such that for all $(u, v) \in \varrho$ and all $a \in \mathcal{A}_R^O(u) \cup \mathcal{A}_S^I(v)$ we have (i) $\mathcal{A}_S^I(v) \subseteq \mathcal{A}_R^I(u)$ (ii) $\mathcal{A}_R^O(u) \subseteq \mathcal{A}_S^O(v)$ (iii) there are $u' \in V_R, v' \in V_S$ such that $(u, a, u') \in \mathcal{T}_R$, $(v, a, v') \in \mathcal{T}_S$ and $(u', v') \in \varrho$.

The control synthesis solution of [4] can be lifted to AIA in the obvious way. The equation under study in our case will be $B \mid M \preceq P$. The usual case is when B is known and M is to be synthesised, but it may also happen that M is given and B is to be synthesised. The solution of [4] can be applied in both cases since the composition of interface automata is commutative. Our methodology is illustrated with the latter case, i.e. we first fix a candidate M derived from P. Then, the synthesis method of [4] is used to obtain B. Our procedure is not always successful: it may be the case that no decomposition is found.

Extracting the adaptation logic. In order to extract a less trivial manager from an AIA P we can proceed as follows. We define the bypassing of an action set $\mathcal{B} \subseteq \mathcal{A}$ in P as $P_{|_{\mathcal{B},\equiv}}$, which is obtained by $P_{|_{\mathcal{B}}}$ (that is, the AIA obtained from P by deleting those transitions whose action belong to $\mathcal{B}$) collapsing the states via the equivalence relation induced by $\{u \equiv v \mid (u, a, v) \in \mathcal{T}_P \wedge a \in \mathcal{B}\}$.

The idea is then to choose a subset $\mathcal{B}$ of $\mathcal{A}_P \setminus \mathcal{A}_P^C$ (i.e. it contains no control action) that the manager M needs not to observe. The candidate manager M is then $P_{|_{\mathcal{B},\equiv}}^{\perp_{\mathcal{A}_P^O \setminus \mathcal{A}_P^C}}$. Of course, if the result is not deterministic, this candidate must be discarded: more observations may be needed.

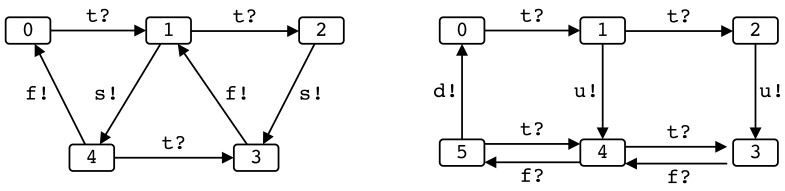

Fig. 10. Bypassed managers for `Scheduler(S)` (left) and `Switch(S)` (right).

Extracting the application logic. We are left with the problem of solving the equation $B \mid M \preceq P$ for given P and M. It is now sufficient to use the solution of [4] which defines B to be $(M \mid P^{\perp})^{\perp}$, where $P^{\perp}$ abbreviates $P^{\perp_{\mathcal{A}_P}}$. If the obtained B and M are compatible, the reverse engineering problem has been solved. Otherwise we are guaranteed that no suitable managed component B exists for the candidate manager M since the solution of [4] is sound and complete. A different choice of control data or hidden actions should be done.

Example 12. The manager $\texttt{Scheduler(S)}^{\perp_{\{\texttt{u},\texttt{d}\}}}_{\{\texttt{u},\texttt{d}\},\equiv}$ (see Fig. 10, left) and the other manager $\texttt{Switch(S)}^{\perp_{\{\texttt{f},\texttt{s}\}}}_{\{\texttt{s}\},\equiv}$ (see Fig. 10, right) are obtained by removing some observations. For the former we obtain no solution, while for the latter we obtain the same base component of the basic decomposition (Fig. 9 left).

5 Related Works

Our proposal for the formalisation of self-adaptive systems takes inspiration by many former works in the literature. Due to lack of space we focus our discussion on the most relevant related works only.

S[B] systems [13] are a model for adaptive systems based on 2-layered transitions systems. The base transition system B defines the ordinary (and adaptable) behaviour of the system, while S is the adaptation manager, which imposes some regions (subsets of states) and transitions between them (adaptations). Further constraints are imposed by S via adaptation invariants. Adaptations are triggered to change region (in case of local deadlock). Weak and strong adaptability formalisations (casted in our setting in Sect. 4.2) are introduced.

Mode automata [12] have been also advocated as a suitable model for adaptive systems. For example, the approach of [20] represents adaptive systems with two layers: *functional layer*, which implements the application logic and is represented by state machines called *adaptable automata*, and *adaptation layer*, which implements the adaptation logic and is represented with a mode automata. Adaptation here is the change of mode. The approach considers three different kinds of specification properties (cf. 4.2): *local*, *adaptation*, and *global*. An extension of linear-time temporal logic (LTL) called mLTL is used to express them.

The most relevant difference between AIA and S[B] system or Mode automata is that our approach does not impose a two-layered asymmetric structure: AIA

can be composed at will, possibly forming towers of adaptation [5] in the spirit of the MAPE-K reference architecture, or mutual adaptation structures. In addition, each component of an adaptive system (be it a manager or a managed component, or both) is represented with the same mathematical object, essentially a well-studied one (i.e. interface automata) decorated with some additional information (i.e. control propositions).

Adaptive Featured Transition Systems (A-FTS) have been introduced in [6] for the purpose of model checking adaptive software (with a focus on software product lines). A-FTS are a sort of transition systems where states are composed by the local state of the system, its configuration (set of active features) and the configuration of the environment. Transitions are decorated with executability conditions that regard the valid configurations. Adaptation corresponds to reconfigurations (changing the system's features). Hence, in terms of our white-box approach, system features play the role of control data. They introduce the notion of *resilience* as the ability of the system to satisfy properties despite of environmental changes (which essentially coincides with the notion of black-box adaptability of [8]). Properties are expressed in AdaCTL, a variant of the computation-tree temporal logic CTL. Contrary to AIAs which are equipped with suitable composition operations, A-FTS are seen in [6] as monolithic systems.

6 Concluding Remarks

We presented a novel approach for the formalisation of self-adaptive systems, which is based on the notion of control propositions (and control actions). Our proposal has been presented by instantiating it to a well-known model for component-based system, interface automata. However, it is amenable to be applied to other foundational formalisms as well. In particular, we would like to verify its suitability for basic specification formalisms of concurrent and distributed systems such as process calculi. Among future works, we envision the investigation of more specific notions of refinement, taking into account the possibility of relating systems with different kind of adaptability and general mechanisms for control synthesis that are able to account also for non-deterministic systems. Furthermore, our formalisation can be the basis to conciliate white- and black-box perspectives adaptation under the same hood, since models of the latter are usually based on variants of transition systems or automata. For instance, control synthesis techniques such as those used to modularize a self-adaptive system (white-box adaptation) or model checking techniques for game models (e.g. [3]) can be used to decide if and to which extent a system is able to adapt so to satisfy its requirements despite of the environment (black-box adaptation).

References

1. de Alfaro, L.: Game models for open systems. In: Dershowitz, N. (ed.) Verification: Theory and Practice. LNCS, vol. 2772, pp. 269–289. Springer (2003)

2. de Alfaro, L., Henzinger, T.A.: Interface automata. In: ESEC/SIGSOFT FSE 2001. ACM SIGSOFT Software Engineering Notes, vol. 26(5), pp. 109–120. ACM (2001)
3. Alur, R., Henzinger, T.A., Kupferman, O.: Alternating-time temporal logic. J. ACM 49(5), 672–713 (2002)
4. Bhaduri, P., Ramesh, S.: Interface synthesis and protocol conversion. Formal Asp. Comput. 20(2), 205–224 (2008)
5. Bruni, R., Corradini, A., Gadducci, F., Lluch-Lafuente, A., Vandin, A.: A conceptual framework for adaptation. In: de Lara, J., Zisman, A. (eds.) FASE. LNCS, vol. 7212, pp. 240–254. Springer (2012)
6. Cordy, M., Classen, A., Heymans, P., Schobbens, P.Y., Legay, A.: Model checking adaptive software with featured transition systems (2012), available at `http://www.info.fundp.ac.be/fts/publications/`
7. De Nicola, R., Vaandrager, F.W.: Action versus state based logics for transition systems. In: Guessarian, I. (ed.) Semantics of Systems of Concurrent Processes. LNCS, vol. 469, pp. 407–419. Springer (1990)
8. Hölzl, M.M., Wirsing, M.: Towards a system model for ensembles. In: Agha, G., Danvy, O., Meseguer, J. (eds.) Formal Modeling: Actors, Open Systems, Biological Systems. LNCS, vol. 7000, pp. 241–261. Springer (2011)
9. IBM Corporation: An Architectural Blueprint for Autonomic Computing (2006)
10. Kulkarni, S.S., Biyani, K.N.: Correctness of component-based adaptation. In: Crnkovic, I., Stafford, J.A., Schmidt, H.W., Wallnau, K.C. (eds.) CBSE. LNCS, vol. 3054, pp. 48–58. Springer (2004)
11. Lints, T.: The essentials in defining adaptation. Aerospace and Electronic Systems 27(1), 37–41 (2012)
12. Maraninchi, F., Rémond, Y.: Mode-automata: About modes and states for reactive systems. In: Hankin, C. (ed.) ESOP. LNCS, vol. 1381, pp. 185–199. Springer (1998)
13. Merelli, E., Paoletti, N., Tesei, L.: A multi-level model for self-adaptive systems. In: Kokash, N., Ravara, A. (eds.) FOCLASA 2012. EPTCS, vol. 91, pp. 112–126 (2012)
14. 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)
15. Robertson, P., Shrobe, H.E., Laddaga, R.: Introduction to self-adaptive software: Applications. In: Robertson, P., Shrobe, H.E., Laddaga, R. (eds.) IWSAS. LNCS, vol. 1936, pp. 1–5. Springer (2001)
16. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Transactions on Autonomous and Adaptive Systems 4(2), 1–42 (2009)
17. Salvaneschi, G., Ghezzi, C., Pradella, M.: Context-oriented programming: A programming paradigm for autonomic systems (v2). CoRR abs/1105.0069 (2012)
18. Zadeh, L.A.: On the definition of adaptivity. Proceedings of the IEEE 51(3), 469–470 (1963)
19. Zhang, J., Goldsby, H., Cheng, B.H.C.: Modular verification of dynamically adaptive systems. In: Moreira, A., Schwanninger, C., Baillargeon, R., Grechanik, M. (eds.) AOSD. pp. 161–172. ACM (2009)
20. Zhao, Y., Ma, D., Li, J., Li, Z.: Model checking of adaptive programs with mode-extended linear temporal logic. In: EASe. pp. 40–48. IEEE Computer Society (2011)