Formal Design and Verification of Self-Adaptive Systems with Decentralized Control

PAOLO ARCAINI, Charles University, Faculty of Mathematics and Physics, Czech Republic
ELVINIA RICCOBENE, Università degli Studi di Milano, Italy
PATRIZIA SCANDURRA, Università degli Studi di Bergamo, Italy

Feedback control loops that monitor and adapt managed parts of a software system are considered crucial for realizing self-adaptation in software systems. The MAPE-K (Monitor-Analyze-Plan-Execute over a shared Knowledge) autonomic control loop is the most influential reference control model for self-adaptive systems. The design of complex distributed self-adaptive systems having decentralized adaptation control by multiple interacting MAPE components is among the major challenges. In particular, formal methods for designing and assuring the functional correctness of the decentralized adaptation logic are highly demanded.

This article presents a framework for formal modeling and analyzing self-adaptive systems. We contribute with a formalism, called *self-adaptive Abstract State Machines*, that exploits the concept of multiagent Abstract State Machines to specify distributed and decentralized adaptation control in terms of MAPE-K control loops, also possible instances of MAPE patterns. We support validation and verification techniques for discovering unexpected interfering MAPE-K loops, and for assuring correctness of MAPE components interaction when performing adaptation.

CCS Concepts: • **General and reference** → *Validation*; *Verification*; • **Software and its engineering** → *System modeling languages;*

Additional Key Words and Phrases: Self-adaptation, MAPE-K loop, MAPE pattern, abstract state machines, formal modeling and analysis, functional requirements assurance

ACM Reference Format:
Paolo Arcaini, Elvinia Riccobene, and Patrizia Scandurra. 2016. Formal design and verification of self-adaptive systems with decentralized control. ACM Trans. Auton. Adapt. Syst. 11, 4, Article 25 (January 2017), 35 pages.
DOI: http://dx.doi.org/10.1145/3019598

1. INTRODUCTION

Self-Adaptation (SA) [Cheng et al. 2009; de Lemos et al. 2013; Kephart and Chess 2003] is an effective approach to deal with the increasing complexity, uncertainty, and dynamicity of modern software systems. These typically operate in dynamic environments and deal with highly changing operational conditions: components can appear

The research reported in this article has been partially supported by the Czech Science Foundation project number 17-12465S, and by the EC within the H2020 under grant agreement 644579 (ESCUDO-CLOUD).
Authors' addresses: P. Arcaini, Faculty of Mathematics and Physics, Charles University, Malostranske namesti, 25, 118 00 Praha 1, Czech Republic; email: arcaini@d3s.mff.cuni.cz; E. Riccobene, Dipartimento di Informatica, Università degli Studi di Milano, via Bramante, 65, 26013 Crema (CR), Italy; email: elvinia.riccobene@unimi.it; P. Scandurra, Dipartimento di Ingegneria Gestionale dell'Informazione e della Produzione, Università degli Studi di Bergamo, viale Marconi, 5, 24044 Dalmine (BG), Italy; email: patrizia.scandurra@unibg.it.

DOI: http://dx.doi.org/10.1145/3019598

and disappear, may become temporarily or permanently unavailable, may change their behavior, etc. A self-adaptive system is able to adapt autonomously to internal dynamics and changing conditions in the environment in order to achieve particular quality goals and to ensure the required functionality.

Feedback control loops that monitor and adapt managed parts of a software system when needed have been identified as crucial elements in realizing self-adaptation of software systems. The MAPE-K (Monitor-Analyze-Plan-Execute over a shared Knowledge) control loop is the most influential reference control model for autonomic and self-adaptive software systems [Kephart and Chess 2003; Brun et al. 2009].

Complex, large, heterogeneous systems may have several *adaptation concerns*, that is, system properties for which adaptation is applied (e.g., optimize performance, handle errors, protect against threats, etc.). A single autonomic loop may not be sufficient for managing all of them. Multiple MAPE-K loops have to be considered and different components may be required to perform the MAPE computations [Weyns et al. 2013]. These components operate in a distributed setting and communicate directly or indirectly by sharing information through a knowledge repository. MAPE computations may be decentralized through the multiple loops and, therefore, have to be coordinated with one another to avoid conflicting situations.

Facing such a complexity is among the major challenges in the field of self-adaptation, and *rigorous methods* to design and reason on distributed self-adaptive systems are highly requested. In particular, there is a demand for formally founded design models that cover both structural and behavioral aspects of self-adaptation, and for approaches to validate and verify behavioral properties and guarantee functional correctness of the adaptation logic. However, although there is an increasing interest in formal methods for SA, the number of studies remains low, and they are mainly related to runtime verification [Weyns et al. 2012].

In this article, we present a conceptual and methodological framework for modeling and verification of distributed self-adaptive systems with decentralized control. We here define a formalism, called *self-adaptive Abstract State Machines*, that exploits the concept of multiagent Abstract State Machines (ASMs) [Börger and Stärk 2003] to specify decentralized adaptation control by using MAPE-K loops. A self-adaptive ASM consists of a set of running agents divided into *managing* ones to control and perform the adaptation logic, and *managed* ones to perform the functional logic. We formalize a MAPE-K control loop in terms of actions of distributed managing agents that communicate directly, or indirectly, by means of knowledge information represented by mathematical functions. Then, we face the problem of giving operational semantics to *MAPE patterns*, that is, patterns identified as recurring structures of interacting MAPE components [Weyns et al. 2013]. To this purpose, we extend our MAPE-K control loop definition in order to model *decentralization* not only *at loop level* among the four MAPE computations, but also *at computation level* among the components interacting to perform the single computation of the loop. Therefore, we are able to express the architecture and the behavior of self-adaptive systems where control decisions are coordinated among different components, regardless of how those control components are physically distributed.

We support formal techniques for validating and verifying ASM models of self-adaptive systems. For validation, we show how to simulate adaptation scenarios and get feedback, at the early stages of the system design, of correct functionality of specified MAPE-K loops. For verification, we describe different approaches, mainly based on static analysis and model checking. In particular, we present a verification technique, based on the proof of metaproperties, useful to discover unwanted interferences between MAPE-K loops. We also provide properties to guarantee correctness of the MAPE components interaction when performing adaptation.

The proposed framework is primarily tailored to the formalization and analysis of functional aspects of adaptation behaviors in order to provide guarantees of correctness of the adaptation [Iftikhar and Weyns 2014]. We do not consider modeling uncertainty, timing aspects, and quality properties. Specifically, we contribute with the following key aspects: (i) definition of a practical (i.e., not requiring particular skills in formal methods), flexible (i.e., adaptable at any desired abstraction/refinement level), executable (i.e., endowed with a simulating virtual machine), well-founded (i.e., based on mathematical definitions) formalism for rigorous modeling; (ii) formalization of adaptation logic in terms of MAPE-K control loops modeled by agents' actions; (iii) decentralized model of computation for control loops; (iv) modular and incremental design process due to separation of concerns (e.g., managing/managed systems, adaptation/functional logics, centralized/decentralized adaptation control); (v) semantics of interaction patterns of MAPE computations; and (vi) verification approaches for checking conflicting computations, pattern instantiations, and adaptation requirements satisfaction. These aspects, as discussed in Section 8.1, are advantages over previous works that do not cover all these features or cover them only partially.

The definition of self-adaptive ASMs and some techniques to analyze these models were already presented in Arcaini et al. [2015]. With respect to the preliminary results presented in that work, the current contribution and (1) extends and improves the formal approach to specify the behavior of a self-adaptive system in terms of decentralized multiple MAPE-K control loops; (2) provides computational semantics to MAPE components (and loops) interaction, and therefore to the notation used to express (instances of) MAPE patterns; (3) presents verification strategies to assure correctness of MAPE components interaction as required by the adaptation logics.

The choice of the ASMs as a formal specification method is based on different considerations: (a) ASMs support the concept of distributed computation, which is fundamental for modeling systems where control is decentralized among multiple distributed components. (b) Although the ASM method comes with a rigorous foundation, the practitioner needs no special training to use it since it can be understood correctly as pseudocode or Virtual Machines working over abstract data structures. (c) ASM models are executable specifications, therefore suitable for high-level model validation, and they are supported by a set of tools for model verification. (d) ASM modeling is based on refinement that permits one to choose the desired level of abstraction when starting to model, and to introduce details along a chain of refined models (each proved to be a correct refinement of the model at the previous level) until, possibly, to code level; this helps in keeping the complexity of the system under control. (e) ASMs permit modeling by specification composition; this is useful to guarantee separation of concerns, in general indispensable for modeling large and complex systems, and in particular for modeling self-adaptive systems.

The article is organized as follows. Section 2 provides some background on the ASM formal method with special emphasis on the multiagent computational model that we exploit for the specification of distributed self-adaptive systems. Section 3 describes the reference model we adopt for realizing SA and presents the ASM-based formalization of a MAPE-K control loop for an adaptation concern; application of our approach is shown on a case study chosen simply for this purpose. Section 4 presents MAPE patterns and the extension of our framework to formalize a MAPE-K control loop where MAPE computations are distributed among multiple components interacting in accordance with a specific schema of communication given as MAPE pattern instance. In Section 5, we present *scenarios* as a validation technique to reproduce and inspect the execution of MAPE-K loops. Section 6 presents verification techniques to check that a system model is of a certain quality (e.g., to identify early knowledge inconsistent updates due to interfering MAPE-K loops) and reflects a desired MAPE pattern. The section also

introduces static and dynamic properties to guarantee correct interaction of MAPE components when performing a control loop. Section 7 presents related work with regard to a number of desired features required for formal methods for modeling and analyzing distributed self-adaptive systems. Section 8 discusses some advantages and disadvantages of our approach. Section 9 concludes the article and outlines future directions.

2. BACKGROUND ON THE ASMs

We here recall the fundamental concepts of the ASM formalism [Börger and Stärk 2003], useful to understand our formalization approach.

(*Basic*) *ASMs* are transition systems that can be viewed as a powerful extension of Finite State Machines, where unstructured control states are replaced by multisorted first-order *structures*, and state transitions are expressed in terms of *rules*.

An *ASM state S* represents an instantaneous system configuration. It is mathematically represented by an algebra with domains of objects and functions defined on them. A pair $(f, (v_1, \ldots, v_n))$ of a function name f, which is fixed by the signature, and a list of dynamic parameter values v_i of whatever type, is called *location*, and it represents the abstract ASM concept of basic object container (or memory unit). Location *updates* represent the basic units of state change and they are given as assignments, each of the form $loc := v$, where *loc* is a location and v its new value.

ASM transition rules describe the system configuration changes, that is, how function interpretations are modified from one state to the next one. The basic form of a transition rule is the *guarded update*: "**if** *Condition* **then** *Updates,*" where *Updates* is a set of function updates of the form $f(t_1, \ldots, t_n) := t$, which are simultaneously executed when *Condition* is true; f is an arbitrary n-ary function and $t_1, \ldots, t_n, t$ are first-order terms. Besides the guarded rule, there is a finite set of *rule constructors* to model simultaneous parallel actions (`par`), nondeterminism (`choose`), unrestricted synchronous parallelism (`forall`), and domain extension (`extend`). Due to their parallel execution, we require updates to be consistent, that is, no pair of updates can simultaneously update the same location to different values.

Functions remaining unchanged during the computation are *static*. Those updated by agent actions are *dynamic*, and distinguished in *monitored* (read by the machine and modified by the environment) and *controlled* (read and written by the machine).

A *computation* of an ASM is a finite or infinite sequence $S_0, S_1, \ldots, S_n, \ldots$ of states of the machine, where S_0 is an initial state and each S_{n+1} is obtained from S_n by simultaneously firing all the transition rules that are enabled in S_n. We denote by machine *step* the state change from a state S_i to the next state S_{i+1}.

The (unique) *main rule* represents the starting point of the computation. An ASM can have more than one *initial state*. It is also possible to specify state *invariants*.

Multiagent ASMs. To support modeling of distributed systems, the notion of *basic-ASMs*, which formalizes simultaneous parallel actions of a *single* agent, has been extended to the notion of *multiagent ASMs* where *multiple* agents act and interact in a synchronous or asynchronous manner. A multiagent ASM is defined as the couple

$$M = \left\langle \{(a, ASM(a)) | a \in Agents\}, main \right\rangle.$$

The first element is a set of pairs where a is an agent of the dynamic finite set *Agents*, and $ASM(a)$ is a machine specifying the agent's behavior (e.g., in Figure 1, the behavior of agent a_3 is specified by $ASM(a_3) = B$). Different agents can have the same behavior and, therefore, the same associated machine. In this case, a special (reserved name) 0-ary function *self* on *Agents*, interpreted by each agent a as itself, is used to denote the agents that are executing the underlying "same" but differently instantiated ASM. Each agent a has a "local" view, $View(a, S)$, of the *global state* S, and it consists of

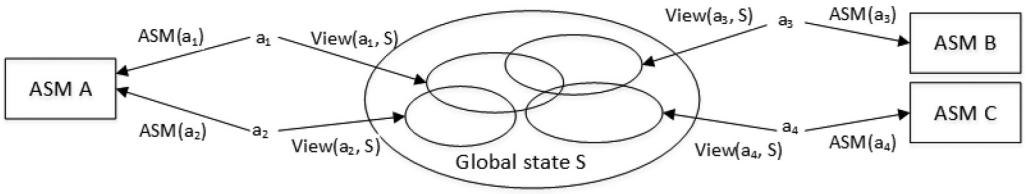

Fig. 1. Multiagent ASM.

the set of locations controlled by the agent a (i.e., locations updated by the machine $ASM(a)$). Agents can have a shared view of a portion of a state by means of *shared* functions, which are used to model communication among parties. Shared functions are dynamic functions that can be updated by the rules of two different agents and can be read by both. Typically, a protocol is needed to guarantee consistency of shared function updates.

The second element in M is the *main rule* that is used to schedule the agents' execution. If all the agents are executed in parallel, the ASM is said to be *synchronous*. Instead, if at each step the main rule selects only a subset of agents to run, the ASM is a particular case of *multiagent asynchronous ASMs* defined in Börger and Stärk [2003]. A run of a multiagent ASM is obtained by the repeated execution of the main rule that computes the new global state: if the ASM is synchronous, all the agents contribute to the new state (by executing their machine $ASM(a)$), otherwise, if the ASM is asynchronous, the new state is obtained by the contribution of a subset of agents.

Tools. The framework ASMETA[1] [Arcaini et al. 2011] provides a set of tools supporting the ASM formal method for model editing, validation, and verification. The tools are strongly integrated in order to permit reusing information about models during several activities. All the specifications of this work were developed using the `AsmetaL` syntax, and validated and verified using the tools provided by the framework.

The framework supports the multiagent ASM computational model. When specifying a multiagent ASM M in `AsmetaL`, the signatures (i.e., definitions of domains and functions) of the composing (basic) ASMs are merged together as signature of M, and a predefined dynamic function *program* on *Agents* is used in the main rule of M to associate an agent a with the main rule of its $ASM(a)$. The function *program* can be used at runtime to dynamically associate or change behavior to agents.

3. A FORMAL FRAMEWORK FOR MODELING SELF-ADAPTATION

Here we present a formal framework, based on the ASM formalism and the ASMETA framework, to model self-adaptive systems in terms of MAPE-K feedback loops for SA. We first recall the typical architecture model of a self-adaptive system with MAPE-K loops, and then we formalize it in terms of ASMs.

3.1. Reference Model for Self-Adaptation

We rely on the reference model for autonomic control loop initially proposed by IBM [Kephart and Chess 2003; Huebscher and McCann 2008] and later shared by many others (as in FORMS [Weyns et al. 2010a]). It is shown in Figure 2. It basically consists of two layers: a *managed subsystem* layer that comprises the application logic, and a *managing subsystem* layer, on top of it, comprising the adaptation logic.

The managing subsystem is conceived as a set of *interacting feedback loops*, one per each self-adaptation aspect (also called *concern* or *goal*). In a feedback loop, the managing system monitors (by reading sensor data) the environment and the managed

[1]http://asmeta.sourceforge.net/.

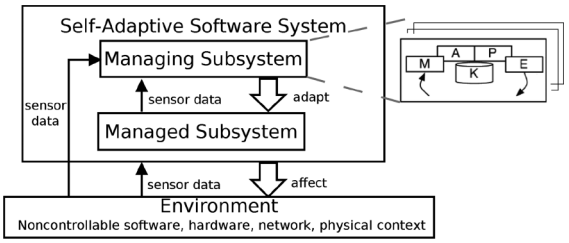

Fig. 2. Reference model for self-adaptive software systems.

subsystem, and adapts the latter when necessary. Therefore, systems are endowed with a self-adaptive layer to support self-* properties (self-healing that allows the system to detect failures and recover autonomously, self-optimizing when operating conditions change, self-reconfiguration when a goal changes, etc. [Huebscher and McCann 2008]) with the intent to improve their quality of service and autonomicity.

A common approach to realize a feedback loop is by means of a MAPE-K loop: A component Knowledge (K) maintains data of the managed system and environment, adaptation goals, and other relevant states that are shared by the MAPE components. A component Monitor (M) gathers data from the managed system and the environment through *probes* (or *sensors*), and saves data in the Knowledge (monitoring). A component Analyze (A) performs data analysis to check whether an adaptation is required (analyzing). If so, it triggers a component Plan (P) that composes a workflow of adaptation actions necessary to achieve the system's goals (planning). These actions are then carried out by a component Execution (E) through *effectors* (or *actuators*) of the managed system (execution).

Other layers can be added to the system where higher-level managing subsystems manage underlying subsystems, which can be managing systems themselves. This managing multilayers architecture leads to the concept of interacting MAPE-K control loops and their recurring organization structures; they are analyzed in Section 4.

3.2. Self-Adaptive ASMs

To model the reference model for SA (see Figure 2) and the operation of MAPE-K control loops, we need to capture all relevant concepts in terms of ASM modeling elements. Since self-adaptive systems have distributed settings, we use the notion of *multiagent ASM*, where *multiple* agents interact in a synchronous or asynchronous way. The computational model of our framework for SA is based on the following definition:

Definition 3.1 (*Self-adaptive ASM*). A *self-adaptive ASM* is a multiagent ASM $M = \langle \{(a, ASM(a)) | a \in Agents)\}, main \rangle$ where the set *Agents* is the disjoint union of the set *MgA* of *managing agents* and the set *MdA* of *managed agents*.

Managing agents encapsulate the logic of self-adaptation, while managed agents encapsulate the system's functional logic. All together these agents comprise the logic of a distributed ASM that models the overall functionality of a self-adaptive system able to monitor the environment and itself and to self-adapt accordingly.

A self-adaptive system may expose certain adaptation concerns $adj_1, \ldots, adj_n$. Note that with adj_i we intend a *type* of adaptation concern, which can have more instantiations at runtime (this commonly happens in adaptive systems, since they are usually composed of different components of the same nature). According to Figure 2, for each (type of) concern *adj*, a MAPE-K loop must be defined.

In the following, we show how (i) a self-adaptive ASM models the structure of the MAPE-K control loop; (ii) the concepts of sensors and actuators are captured in the

ASM model, and MAPE computations are implemented in terms of ASM rules; (iii) a MAPE-K control loop establishes an (direct or indirect) interaction relation among the MAPE components and is executed in a centralized or decentralized way; and (iii) a self-adaptive ASM executes by means of the cooperation and coordination of different (managing and managed) agents. In this section, we consider *basic* MAPE-K loops, that is, loops composed of only four computations.[2] More complex loops, in which the same computations can be distributed among different agents, will be treated in Section 4 when MAPE interaction patterns are considered.

Modeling MAPE-K Loop Structure. In ASMs, computations are naturally expressed in terms of transition rules, since rules are able to represent decisions (by rule guards) and actions (by update rules). Therefore, for each (type of) adaptation concern adj, the self-adaptive ASM models the computations of a MAPE-K control loop *MAPE-K*(adj) by means of a set of ASM transition rules:

$$R_{adj} = \{r^{a_M}_{M\_adj}, r^{a_A}_{A\_adj}, r^{a_P}_{P\_adj}, r^{a_E}_{E\_adj}\}, \tag{1}$$

where a_M, a_A, a_P, and a_E are the (types of) managing agents that are involved in the adaptation concern adj.

$r^{a_X}_{X\_adj}$ (where X stands for M, A, P, or E) is the ASM rule modeling the computation X of $MAPE - K(adj)$, performed by the agent (type) a_X. Note that the same agent (type) may be involved in different computational parts of the same control loop (i.e., a_M, a_A, a_P, and a_E are not necessarily distinct). We also like to remark that, depending on the level of abstraction chosen to model a control loop, a rule $r^{a_X}_{X\_adj}$ may specify together two subsequent MAPE computations as a unique activity. A suitable model refinement may expose the separation of the two computations.

We annotate (as comments //) rules involved in a loop *MAPE-K*(adj) with labels @M_adj (for monitoring), @A_adj (for analyzing), @P_adj (for planning), and @E_adj (for execution); in case a rule models two consecutive computations X and Y (with $X \in \{M, A, P\}$ and $Y \in \{A, P, E\}$), we use the annotation @XY_adj. From a specification point of view, these labels help in understanding how the MAPE computations are distributed among managing agents; for verification purposes, they help to automatically extract from the model specific information (see Section 6.2 for more details).

Rules in the set R_{adj} as in (1) reason over the knowledge $K(adj)$, which is naturally modeled by means of functions, since in ASMs system memory is represented in terms of functions. $K(adj)$ corresponds to that part of the signature of the self-adaptive ASM used to represent the managed subsystem and the environment, and other information for the enactment and coordination of $MAPE - K(adj)$ computations.

Rules in R_{adj} are not independent of each other, but a specific execution order exists that is expressed by the following *interaction relation*:

$$\xrightarrow{adj} \subset R_{adj} \times R_{adj},$$

where $r_X \xrightarrow{adj} r_Y$ means that r_X produces some information in $K(adj)$ that is used (i.e., read) by r_Y. This binary relation reflects the fact that in a MAPE-K control loop a monitoring computation interacts with an analyze computation, which, in turn, interacts with a planning computation, which interacts with an execution computation. In case of a not complete loop, some computations can be skipped, but the order is kept. Therefore, on the set R_{adj} in (1), it yields

$$\xrightarrow{adj} = \{(r^{a_M}_{M\_adj}, r^{a_A}_{A\_adj}), (r^{a_A}_{A\_adj}, r^{a_P}_{P\_adj}), (r^{a_P}_{P\_adj}, r^{a_E}_{E\_adj})\}.$$

[2]Note that we here use the term *computation* to denote the activity performed by the MAPE components.

In conclusion, *MAPE – K*(*adj*) is completely defined by a set of transition rules, their interaction relation, and the knowledge as follows:

$$MAPE - K(adj) = \langle R_{adj}, \xrightarrow{adj}, K(adj) \rangle. \quad (2)$$

MAPE-K Rules Implementation. The notion of *environment* is directly supported in the ASM theory by means of ASM monitored functions, that is, locations read by the machine and modified by the environment (by means of an *oracle*). The four rules composing the MAPE-K loop exploit the environment, as well as the knowledge, as follows:

—r_M models *context-aware monitoring*, that is, perceiving the state of the environment, and *self-aware monitoring*, that is, perceiving the state of the managed (and, if necessary, of the managing itself) subsystem. *Probes* (or *sensors*) from the environment for context-aware monitoring are modeled as ASM monitored functions or derived functions defined in terms of monitored functions; probes from the managed system for *self-aware monitoring* are represented by *shared* functions updated by the managed agent and only read by the managing agent.
—r_A checks if adaptation is necessary and so if an adaptation plan has to be triggered.
—r_P creates or selects a procedure to enact a necessary adaptation in the managed system. It can be a single action or a complex workflow.
—r_E carries out the adaptation actions on the managed system as decided by the planning, by using actuators.

Actuators (or *effectors*) are specified in terms of actions (ASM rules) of managed agents that update own controlled locations according to the adaptation plan decided by managing agents. Actuators are indirectly triggered by execution computations (performed by managing agents) by updating locations shared with the managed system. The following atomic actions are supported as adaptation operators:
—change locations shared between the managed agent and the managing agent to (indirectly) trigger the execution of the actuators;
—stop/start managed agents by setting their *program* to the *skip*-rule and a given rule r, respectively;
—dynamically instantiate a new managed agent to introduce a new concurrent behavior by an *extend*-rule over the domain *MdA* of managed agents;
—dynamically change the behavior of a managed agent a by updating *program*(a) to a new rule r.
Note that, in the case of multi managing layers, these adaptation actions can be executed in a reflective manner to adapt the managing layer itself.

Computation Models of MAPE-k Loops. The control loop formalization as in (2) statically identifies the (types of) agents involved in the loop execution, models their roles in the adaptation, and their interaction relation. Therefore, it specifies *what* pieces of the agent's behavior are involved in a MAPE-K loop, but not *how* the loop is executed and, in particular, how a MAPE computation interacts with the subsequent one. There are two schemes of execution: *decentralized* and *centralized*.

In the decentralized scheme, the four rules in *MAPE – K*(*adj*) are executed in different run steps by different agents, which interact with each other *indirectly* through the knowledge $K(adj)$. More precisely, if r_{X_adj} is in interaction relation with r_{Y_adj} (i.e., $r_{X\_adj} \xrightarrow{adj} r_{Y\_adj}$), when agent a_X executes rule r_{X_adj}, it can update some locations of $K(adj)$ that agent a_Y can read when it executes rule r_{Y_adj}.

In the centralized scheme, rules in R_{adj} are executed by the same managing agent (i.e., a_M, a_A, a_P, and a_E in (1) identify the same agent). Each rule r_{X_adj} can interact with the rule r_{Y_adj} (i.e., the rule it is in interaction relation with) either *indirectly* (as in the decentralized scheme) or *directly*. In the direct interaction, rule r_{X_adj} calls rule r_{Y_adj} and, therefore, the two rules are executed in one step. Consequently, if each rule of a MAPE-K loop directly calls the rule it is in interaction relation with, all the computations of the loop are executed in one step.

A mixed scheme of direct and indirect interaction is also possible; for example, monitoring and analysis computations may be executed by a given agent (i.e., $a_M = a_A$) through a direct interaction and, in a subsequent step (indirect interaction), a (possibly different) agent may execute the planning and execute computations (i.e., $a_P = a_E$) using the direct interaction scheme.

Self-adaptive ASM Execution. The execution of a self-adaptive ASM is the synchronous or asynchronous execution of all agents in $Agents = MgA \cup MdA$ as specified by the *main* rule. The predefined dynamic function *program* on *Agents* indicates the (main rule of the) ASM associated with an agent. Programs of managed agents are any kind of ASMs, while programs of managing agents contain rules annotated by $@X$ ($X = M, A, P, E$), since they are responsible for performing MAPE computations. Note that, since a managing agent $a_j \in MgA$ may be involved in one or more loops $MAPE-K(adj_h)$, $h = j_1, \ldots, j_k$, its program $program(a_j)$ is a suitable coordination, by means of conventional ASM rule constructors (par, if-then-else, etc.), of rules $r^{a_j}_{X\_adj_h}$ modeling the behavioral contribution of the agent to the loops $adj_{j_1}, \ldots, adj_{j_k}$ it is involved in.

3.3. Smart Home Case Study

In order to show the application of self-adaptive ASMs, we consider the *smart home* case study proposed in Song et al. [2013]. In a smart home, different sensors are used to collect different data (e.g., temperature, humidity, etc.) that are analyzed by special devices that can make some decisions and modify the house components (e.g., appliances, windows, heating system, etc.) accordingly. In this way, the house is *smart*, as it is able to automatically adapt to the changing of environment conditions, in order to improve living qualities and to save resources. We consider the five adaptation concerns reported in Song et al. [2013]:

(1) *Comfortable heating* (*CH*): when it is cold, the heaters should be at sufficient settings for comfort;
(2) *Minimize dispersion* (*MD*): do not open window when the heater is on;
(3) *Air quality* (*AQ*): do open window when air quality is bad;
(4) *Morning water heating* (*MWH*): keep water heater on in the early morning;
(5) *Electricity saving* (*ES*): when the electricity is expensive, do not use water heater and strong heating together.

We propose a self-adaptive ASM[3] whose instantiation of the reference model is shown in Figure 3: there are *five* managing agents involved in the control loops, *three* managed agents, and a different MAPE-K control loop for each adaptation concern.[4]

Managed agents of type `HeaterManaged`, `WindowManaged`, and `WaterHeaterManaged` model the normal behavior of the house, namely, of the windows, the heaters, and the water heater. They usually modify their status when triggered by the MAPE-K loops.

[3]All the specifications are available online at http://fmse.di.unimi.it/sw/TAAS2017.zip.
[4]For simplicity of the presentation, we suppose to have only one room with one window and one heater, but multiple rooms with multiple windows and heaters can be modeled.

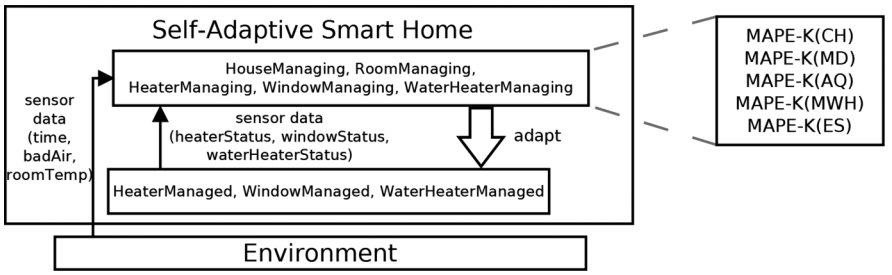

Fig. 3. Self-adaptive smart home.

Two kinds of managing agent are responsible for the monitoring and data analysis: an agent of type `RoomManaging` monitors the temperature and the air quality of a room, and an agent of type `HouseManaging` monitors the water heater and the heating consumption of the whole house.

Three other kinds of managing agent are instead responsible for triggering the adaptation of the house components to the environment changing conditions. Each agent of type `HeaterManaging` can trigger the modification of the heater status; each agent of type `WindowManaging` can trigger the opening/closing of a window; each agent of type `WaterHeaterManaging` can trigger the turning on/off of the water heater. Such status modification will become effective upon firing the adapters from the responsible managed agents. Code 1 shows the definitions and the schedule of the agents' programs.

Adaptation strategies of the smart home system are specified by simple control loops that have no real plan computations since, resembling Event-Condition-Action (ECA) rules, adaptation can be executed directly upon the analysis of specific sensor data combinations. Although the structure of such MAPE-K loops is simple, it is useful and effective to illustrate clearly the formal framework.

We here show the definitions of the first four loops (see Equation 2). The fifth loop will be described in Section 4.1, since it requires a more advanced definition of loop (in terms of MAPE pattern).

$$MAPE-K(CH) = \langle R_{CH}, \xrightarrow{CH}, K(CH) \rangle, \qquad MAPE-K(MD) = \langle R_{MD}, \xrightarrow{MD}, K(MD) \rangle,$$
$$MAPE-K(AQ) = \langle R_{AQ}, \xrightarrow{AQ}, K(AQ) \rangle, \qquad MAPE-K(MWH) = \langle R_{MWH}, \xrightarrow{MWH}, K(MWH) \rangle,$$

where

R_{CH} = {r_checkRoomTempMAPE_CH, r_adaptHeaterMAPE_CH}
$\xrightarrow{CH}$ = {(r_checkRoomTempMAPE_CH, r_adaptHeaterMAPE_CH)}
$K(CH)$ = {roomTemp, sgnHeaterVERY_HOT_CH, sgnHeaterFAIRLY_HOT_CH, sgnHeaterOFF_CH}

R_{MD} = {r_checkWindowAndHeaterMAPE_MD, r_adaptWindowMAPE_MD}
$\xrightarrow{MD}$ = {(r_checkWindowAndHeaterMAPE_MD, r_adaptWindowMAPE_MD)}
$K(MD)$ = {heaterStatus, windowStatus, sgnCloseWindow_MD}

R_{AQ} = {r_checkAirQualityMAPE_AQ, r_adaptWindowMAPE_AQ}
$\xrightarrow{AQ}$ = {(r_checkAirQualityMAPE_AQ, r_adaptWindowMAPE_AQ)}
$K(AQ)$ = {badAir, sgnOpenWindow_AQ, sgnCloseWindow_AQ}

R_{MWH} = {r_checkHotWaterMorningMAPE_MWH, r_adaptWaterHeaterMAPE_MWH}
$\xrightarrow{MWH}$ = {(r_checkHotWaterMorningMAPE_MWH, r_adaptWaterHeaterMAPE_MWH)}
$K(MWH)$ = {time, sgnWaterHeaterON_MWH}

```
asm smartHome
signature:
    //managed agents types
    domain HeaterManaged subsetof Agent
    domain WindowManaged subsetof Agent
    domain WaterHeaterManaged subsetof Agent
    //managing agents types
    domain HouseManaging subsetof Agent
    domain RoomManaging subsetof Agent
    domain HeaterManaging subsetof Agent
    domain WindowManaging subsetof Agent
    domain WaterHeaterManaging subsetof Agent
    //managed agents
    static heaterManaged: HeaterManaged
    static windowManaged: WindowManaged
    static waterHeaterManaged: WaterHeaterManaged
    //managing agents
    static houseManaging: HouseManaging
    static roomManaging: RoomManaging
    static heaterManaging: HeaterManaging
    static windowManaging: WindowManaging
    static waterHeaterManaging: WaterHeaterManaging
    ...
    //Knowledge – sensors
    monitored badAir: Boolean
    monitored roomTemp: Temperature
    //Knowledge – signals
    controlled sgnHeaterOFF_CH: Boolean
    controlled sgnHeaterFAIRLY_HOT_CH: Boolean
    //Knowledge – statuses of managed agents
    controlled heaterStatus: HeaterStatus
    controlled windowStatus: WindowStatus
    controlled waterHeaterStatus: WaterHeaterStatus
    ...
    //shared between managed and managing agents
    controlled setWaterHeaterStatus: WaterHeaterStatus
    controlled setHeaterStatus: HeaterStatus
    controlled setWindowStatus: WindowStatus
definitions:
    ...
    macro rule r_heater = ...
    macro rule r_window = ...
    macro rule r_waterHeater = ...
    macro rule r_monitorAndAnalyzeHouse = ...
    macro rule r_monitorAndAnalyzeRoom = ...
    macro rule r_adaptHeater = ...
    macro rule r_adaptWindow = ...
    macro rule r_adaptWaterHeater = ...

    main rule r_Main =
        if checkForAdaptation then
            par
                program(houseManaging)
                program(roomManaging)
                program(heaterManaging)
                program(windowManaging)
                program(waterHeaterManaging)
                ...
            endpar
        else
            par
                program(heaterManaged)
                program(windowManaged)
                program(waterHeaterManaged)
            endpar
        endif

default init s0:
    //statuses of the managed elements
    function heaterStatus = OFF
    function windowStatus = OPEN
    function waterHeaterStatus = WE_OFF
    //signals
    function sgnHeaterOFF_CH = false
    function sgnHeaterFAIRLY_HOT_CH = false
    function sgnOpenWindow_AQ = false
    function sgnWaterHeaterON_MWH = false
    ...

    agent HeaterManaged: r_heater[]
    agent WindowManaged: r_window[]
    agent WaterHeaterManaged: r_waterHeater[]
    agent HouseManaging: r_monitorAndAnalyzeHouse[]
    agent RoomManaging: r_monitorAndAnalyzeRoom[]
    agent HeaterManaging: r_adaptHeater[]
    agent WindowManaging: r_adaptWindow[]
    agent WaterHeaterManaging: r_adaptWaterHeater[]
```

Code 1. Smart home case study—Managing agents and knowledge.

```
macro rule r_monitorAndAnalyzeRoom =
    par
        if monMAPE_CH then
            r_checkRoomTempMAPE_CH[] //@MA_MAPE_CH
        endif
        if monMAPE_MD then
            r_checkWindowAndHeaterMAPE_MD[] //@MA_MAPE_MD
        endif
        if monMAPE_AQ then
            r_checkAirQualityMAPE_AQ[] //@MA_MAPE_AQ
        endif
    endpar
```

```
macro rule r_checkRoomTempMAPE_CH =
    if roomTemp < 10 then
        sgnHeaterVERY_HOT_CH := true
    else
        if roomTemp < 18 then
            sgnHeaterFAIRLY_HOT_CH := true
        else
            sgnHeaterOFF_CH := true
        endif
    endif
```

Code 2. Smart home case study—Program of agent `RoomManaging` and rule executing MA of *MAPE* – *K*(*CH*).

The managing agent `RoomManaging` is involved in more than one MAPE-K loop since it executes the monitoring and the analysis of *MAPE* – *K*(*CH*), *MAPE* – *K*(*MD*), and *MAPE* – *K*(*AQ*). Code 2 shows the agent's program (i.e., rule `r_monitorAndAnalyzeRoom`).

For *MAPE* – *K*(*CH*), the agent checks the temperature of the room (by means of rule `r_checkRoomTempMAPE_CH` shown in Code 2) and notifies the agent managing the room heater on how to set the heating. Note that all functions having prefix `sgn` are used to

```
macro rule r_adaptHeater =
    par
        if execMAPE_CH then
            r_adaptHeaterMAPE_CH[] //@E_MAPE_CH
        endif
        if execMAPE_ES then
            r_adaptHeaterMAPE_ES[] //@E_MAPE_ES
        endif
    endpar
```

```
macro rule r_adaptHeaterMAPE_CH =
    par
        if sgnHeaterVERY_HOT_CH then
            par
                setHeaterStatus := VERY_HOT
                sgnHeaterVERY_HOT_CH := false
            endpar
        endif
        ...
    endpar
```

Code 3. Smart home case study – Program of agent `HeaterManaging` and rule executing E of *MAPE* – *K*(*CH*).

```
macro rule r_heater =
    if isDef(setHeaterStatus) then
        par
            heaterStatus := setHeaterStatus //actuator
            setHeaterStatus := undef
        endpar
    endif
```

Code 4. Smart home case study – Program of agent `HeaterManaging`.

manage the communication from the managing agents responsible for monitoring and analysis and the managing agents responsible for the execution.

For *MAPE* – *K*(*MD*), the agent checks (by means of rule `r_checkWindowAndHeater-MAPE_MD`) that the window is closed when the heating system is turned on, and, if this is not the case, notifies the window managing agent that the window must to be closed.

For *MAPE* – *K*(*AQ*), the agent checks (by means of rule `r_checkAirQualityMAPE_AQ`) the air quality and notifies the window managing agent that the window has to be opened/closed. For explanation purposes, in the example we have made explicit the condition that guards the execution of each monitoring rule (i.e., `monMAPE_`*X* for the adaptation concern *X*). However, such conditions can have any form and rules can be nested at any depth.

All the MAPE-K loops are executed following a decentralized schema in which the monitoring and analyzing are executed by an agent in the same rule, and the execution is performed by another agent. Let us consider *MAPE* – *K*(*CH*). As seen before, the `RoomManaging` agent is responsible for the monitoring and analyzing computation of this loop. The execution computation is done by an agent of type `HeaterManaging` that reads the signals `sgnHeaterVERY_HOT_CH`, `sgnHeaterFAIRLY_HOT_CH`, `sgnHeaterOFF_CH` (sent by the `RoomManaging` agent), and `sgnHeaterFAIRLY_HOT_ES` (sent by the `HouseManaging` agent), and modifies the status of the heater accordingly. The program of the agent and the rule of the execution computation of *MAPE* – *K*(*CH*) are shown in Code 3.

The adaptation is carried out when the managed agents, triggered by the execution computations of managing agents, adapt their statuses through the actuators. For example, Code 4 shows the program of managed agent `HeaterManaged` that modifies its status whenever notified (i.e., when `setHeaterStatus` is defined) by managing agents `RoomManaging` or `HouseManaging`. Note that all functions having prefix `set` are used to manage the communication from managing to managed agents.

4. INTERACTING MAPE COMPONENTS AND PATTERNS

In Section 3.2, we have modeled each computation M, A, P, and E by means of a single rule executed by a given agent. However, computations M, A, P, and E may be made by multiple components, that is, they may be decentralized throughout interacting groups of MAPE components. To make the interactions of control loops explicit and expose how these interactions are handled, in Weyns et al. [2013] recurring structures of interacting

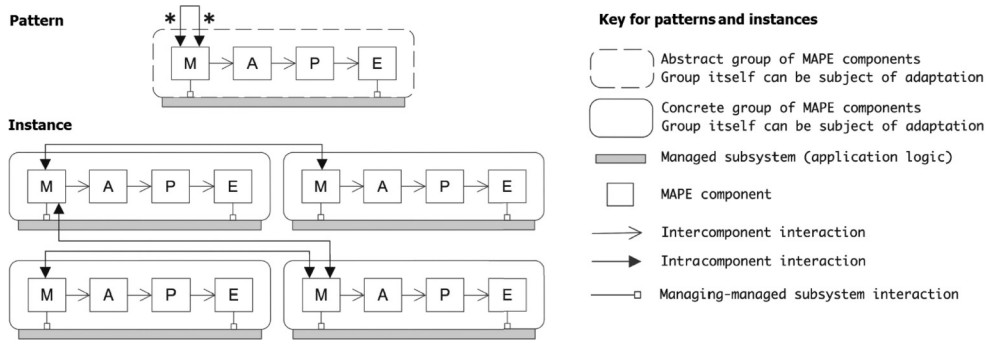

Fig. 4. Information sharing: pattern and concrete instance of the pattern.

MAPE components, defined as *MAPE patterns*, are presented through a graphical notation. A MAPE pattern describes the abstract groups of MAPE components, the type of interactions between MAPE components and between groups, and the interactions with the managed subsystem. A pattern instance describes the concrete structure of the pattern for one particular configuration.

As an example, Figure 4 shows the pattern *information sharing*, an instance of this pattern, and the graphical notation adopted in Weyns et al. [2013]. The annotated cardinalities of the interactions between the groups of MAPE components determine the allowed occurrences of the different groups in the pattern.

In terms of notation, there are different types of interactions:

—*Managed-managing subsystem interactions* are those between M components and the managed subsystem for monitoring purposes, and between E components and the managed subsystem for performing adaptations.
—*Intercomponent interactions* are those between MAPE components of different types. In a typical MAPE-K loop, M interacts with A, A with P, and P with E, but more complex interaction loops are possible [Vromant et al. 2011].
—*Intracomponent interactions* are those between MAPE components of the same type, for example, interactions between M components in the pattern information sharing.

The formalization in Section 3.2 already captures managed-managing interactions by ASM rules of type r_M and r_E, and provides a way to model (inter/intra, direct/indirect) components interaction. Our goal here is to provide an operational semantics (in terms of ASMs) to MAPE patterns defined in Weyns et al. [2013]. In order to reflect the fact that each MAPE computation can be performed by a set of rules (and no more by only one rule), in (1) we extend the set R_{adj} of transition rules as follows:

$$R_{adj} = \left\{ r^{a_{M_1}}_{M_1\_adj}, r^{a_{A_1}}_{A_1\_adj}, r^{a_{P_1}}_{P_1\_adj}, r^{a_{E_1}}_{E_1\_adj}, \ldots, r^{a_{M_n}}_{M_n\_adj}, r^{a_{A_n}}_{A_n\_adj}, r^{a_{P_n}}_{P_n\_adj}, r^{a_{E_n}}_{E_n\_adj} \right\}, \tag{3}$$

where different rules $r^{a_{X_1}}_{X_1\_adj} \ldots r^{a_{X_n}}_{X_n\_adj}$ contribute (through intracomponent interaction) to a given computation X (with $X \in \{M, A, P, E\}$).

The interaction relation $\xrightarrow{adj}$ on R_{adj} reflects a *coordination model*[5] among MAPE computations established at design time when the MAPE-K control loop is designed as interacting groups of MAPE components, possibly by instantiating a MAPE pattern.

[5]According to the classification of coordination models given in Papadopoulos and Arbab [1998], we here intend a data-driven approach where communication is done through the knowledge as *shared dataspace*.

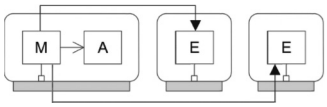

Fig. 5. Smart home case study—Pattern with multiple E components.

Therefore, $r^{a_X}_{X\_adj} \xrightarrow{adj} r^{a_Y}_{Y\_adj}$ if and only if there exists an interaction (inter or intra) from the MAPE component X to the MAPE component Y in the coordination model. Here, the interaction relation both expresses inter- and intracomponent interactions: an interaction relation between rules $r^{a_X}_{X\_adj}$ and $r^{a_Y}_{Y\_adj}$ models an intracomponent interaction if X and Y are of the same type, or an intercomponent interaction otherwise.

As a final remark, we observe that our formalization is also able to specify MAPE-K control loops resulting from the composition of a number of MAPE-K (sub-)loops (in this case, a group of MAPE components performs a control loop). Indeed, if an adaptation concern adj is composed of subconcerns adj_i $(i = 1, \ldots, h)$ whose control (sub-)loops are $MAPE - K(adj_i) = \langle R_{adj_i}, \xrightarrow{adj_i}, K(adj_i)\rangle$, then in (2) it yields

—$R_{adj} = \bigcup_{i=1}^{h} R_{adj_i}$,
—$\xrightarrow{adj} = \bigcup_{i=1}^{h} \xrightarrow{adj_i}$ plus all those (inter/intra) interactions among MAPE computations of different control (sub-)loops established to compose the subloops, and
—$K(adj) = \bigcup_{i=1}^{h} K(adj_i)$.

An example of the formalization of a composed MAPE-K control loop is given in Section 4.1 for the flexibility concern of the traffic monitoring case study.

4.1. Application Examples

Smart Home. The loop ES of the smart home case study (related to electricity saving) is a simple example of a pattern where two rules perform the execution computation (as shown in Figure 5). The loop is defined as follows:

$$MAPE - K(ES) = \left\langle R_{ES}, \xrightarrow{ES}, K(ES)\right\rangle,$$

where

R_{ES} = {r_checkElectrMAPE_ES, r_adaptHeaterMAPE_ES, r_adaptWaterHeaterMAPE_ES}
$\xrightarrow{ES}$ = {(r_checkElectrMAPE_ES, r_adaptHeaterMAPE_ES), (r_checkElectrMAPE_ES, r_adaptWaterHeaterMAPE_ES)}
$K(ES)$ = {time, heaterStatus, waterHeaterStatus, sgnHeaterFAIRLY_HOT_ES, sgnWaterHeaterOFF_ES}

The monitor and analysis components are modeled by the rule `r_checkElectrMAPE_ES` of `HouseManaging` that checks whether both strong heating and water heater are used together. If this is the case, the rule indirectly interacts with either rule `r_adaptHeaterMAPE_ES` of `HeaterManaging` that reduces the speed of the heater, or rule `r_adaptWaterHeaterMAPE_ES` of `WaterHeaterManaging` that turns the water heater off. Requirements in Song et al. [2013] do not specify which action should be taken when adaptation is needed; therefore, at this level of abstraction, we nondeterministically choose one of the two actions. In further refinement steps, a more detailed planning policy could be added for deciding which adaptation action to apply.

Traffic Monitoring. The specification of interacting groups of MAPE components has been mainly experimented in modeling a traffic monitoring application [Iftikhar and Weyns 2012], where a number of intelligent cameras are located along a road to detect traffic jams and collaborate in case of congestion to assist, for example, traffic

light controllers or driver assistance systems. Traffic jams can span the viewing range of multiple cameras and can dynamically grow and dissolve. Cameras are endowed with a data processing unit and a communication unit to interact with each other. They have to collaborate by aggregating their monitored data in the case of traffic congestion because each camera has a limited viewing range. In order to avoid the bottleneck of a centralized control center, the task of the cameras has to be performed in a decentralized way. To this purpose, multiple cameras collaborate in organizations when a traffic jam spans their viewing range. Cameras enter or leave the organization whenever the traffic jam enters or leaves their viewing range.

We already used this case study in Arcaini et al. [2015] where we kept the same adaptation functional requirements and system reference model as in Iftikhar and Weyns [2012]. Here we use a fragment of such a specification to show a concrete example of composed control loops that realize the MAPE pattern information sharing.

The managed system of the reference model for SA are the *cameras*, each endowed with mechanisms to detect traffic jams and inform clients; the managing system consists, for each camera, of two components: the *organization middleware* and *the self-healing* subsystems. Organization middlewares collaborate to manage an organization that spans multiple cameras when congestion is detected; a *master/slave* control model is used for synchronization issues. For each camera, the self-healing subsystem detects failures of other cameras (silent nodes) by using a *Heartbeat* interaction pattern. Therefore, the case study exposes two main adaptation concerns:

—*Flexibility*: two or more cameras start an organization in case of traffic congestion. For each organization, one camera is elected as master, while the others are slaves. A camera leaves the organization when it does not observe congestion anymore.
—*Robustness*: a camera fails and becomes unresponsive without sending data, either in case of an external camera becoming silent (external failure), and in case of internal failure of a camera.

Through interactions among peer M components of the cameras, the overall system realizes the architectural MAPE pattern *information sharing* for the flexibility concern and the robustness concern due to external failures. For the flexibility concern, monitor components, that is, M computations executed by the organization middleware managing systems (one per each camera), interact for restructuring master/slave organizations in case of congestion. For the robustness concern due to external failures, a monitor computation executed by the self-healing managing system (one per each camera) gathers the required data locally from the managed camera, but it also requires coordination with monitor computations executed by other self-healing subsystems to gather the status of other cameras and, eventually, recover from a silent node failure.

Due to the complexity of the case study, we have to reason in terms of the camera's roles, subconcerns, and subloops. Let us consider, for example, the flexibility concern. The role of a camera at a given time can be master of a single member organization, master of an organization with slaves, or slave in an organization. Camera's role determines the behavior of the camera's organization middleware that assumes the same role. For example, in the role of master of a single member organization (the default role of a camera), we can consider three flexibility subconcerns capturing the phases of the master-slave election protocol[6]:

[6]To keep the master election policy simple, we assume every camera has a unique ID that is a monotonically increasing function on the traffic direction and the camera with the lowest ID becomes master. Algorithms to elect a new master, like Bully [Garcia-Molina 1982] and Ring algorithms, are out of the scope of this article.

```
macro rule r_masterBehaviour($c in Camera) =
    par
        r_detectCongestion[$c] //@M_MAPE_CD
        r_analyzeCongestion[$c]
    endpar

macro rule r_detectCongestion($c in Camera) =
    if not(stopCam($c)) and cong($c) and
                                    not congested(self) then
        par
            congested(self) := true
            r_send_s_offer_message[$c]
        endpar
    endif
```

```
macro rule r_analyzeCongestion($c in Camera) =
    par
        r_analyzeCongestionTMS[$c] //@MA_MAPE_TMS
        r_analyzeCongestionTS[$c] //@MA_MAPE_TS
    endpar

macro rule r_analyzeCongestionTMS($c in Camera) =
    if congested(self) and m_offer($c) then
        r_turnMasterWithSlaves[$c] //@PE_MAPE_TMS
    endif

macro r_analyzeCongestionTS($c in Camera) =
    if congested(self) and not(m_offer($c)) and s_offer($c) then
        r_joinOrganization[$c] //@PE_MAPE_TS
    endif
```

Code 5. Rule `r_masterBehaviour`.

(1) *Congestion detection* (*CD*): start negotiations with the next alive camera when congestion is detected for the first time;
(2) *Turning master with slaves* (*TMS*): become master with slaves when the next alive camera accepts to become slave in a congested situation;
(3) *Turning slave* (*TS*): join the organization of the requester camera as slave when receiving the request to become slave in a congested situation.

Code 5 shows the rule `r_masterBehaviour` executed by the organization controller (the ASM managing agent representing the camera's organization middleware) of a camera in the role of master. Details on the macro subrules not reported here can be found in the ASM specification available online. For each subconcern, we define a MAPE-K subloop as follows (see Equation 2):

$$MAPE-K(CD) = \langle R_{CD}, \xrightarrow{CD}, K(CD)\rangle, \quad MAPE-K(TMS) = \langle R_{TMS}, \xrightarrow{TMS}, K(TMS)\rangle,$$
$$MAPE-K(TS) = \langle R_{TS}, \xrightarrow{TS}, K(TS)\rangle,$$

where

$R_{CD} = \{r_{M_{CD}} \equiv$ r_detectCongestion$\}$ $\quad \xrightarrow{CD} = \varnothing \quad K(CD) =$ {congested,s_offer}

$R_{TMS} = \{r_{MA_{TMS}} \equiv$ r_analyzeCongestionTMS, $r_{PE_{TMS}} \equiv$ r_turnMasterWithSlaves$\}$

$\xrightarrow{TMS} = \{(r_{MA_{TMS}}, r_{PE_{TMS}})\} \quad K(TMS) =$ {congested,m_offer,state,slaves,newSlave}

$R_{TS} = \{r_{MA_{TS}} \equiv$ r_analyzeCongestionTS, $r_{PE_{TS}} \equiv$ r_joinOrganization$\}$

$\xrightarrow{TS} = \{(r_{MA_{TS}}, r_{PE_{TS}})\} \quad K(TS) =$ {congested,s_offer,newSlave,state,change_master,getMaster}

The first subloop $MAPE-K(CD)$ is a monitor computation. It starts negotiations when congestion is detected for the first time (the monitored predicate *cong* is true and the controlled predicate *congested* is false): the organization controller sends a request to join its organization as slave (the knowledge location *s_offer*) to the organization controller of the next alive camera (if any) in the direction of the traffic.

The second subloop $MAPE-K(TMS)$ deals with the second part of the master-slave election protocol in case of congestion. After inviting the next alive camera to join the organization as slave (the knowledge location *s_offer*), the organization controller waits for receiving the signal *m_offer* back from the invited camera as positive (acceptance) answer. When it receives such a signal, it becomes MASTER of the joined organization by executing the rule `r_turnMasterWithSlaves` to concretely add the new slave to its list and change the camera role in "master with slaves."

The third subloop *MAPE – K*(*TS*) captures the counterpart behavior of the master-slave protocol. In a congested situation, if the organization controller receives the request to become slave (the knowledge location *s_offer*), it joins the organization of the requester camera as slave by directly executing the rule `r_joinOrganization` that sends the signal *m_offer* back to the requester and turns the camera in the role of slave.

The coordination between the first subloop and the other two subloops (see the rule `r_masterBehaviour` in Code 5) is realized by a par-construct, but a sequential execution is guaranteed by the rules conditions that are mutually exclusive. Moreover, the third subloop may be executed (see the rule `r_analyzeCongestion` in Code 5) only if the triggering condition of the second subloop is false (alternate logic) to give priority to *m_offer* signals with regard to *s_offer* signals thus avoiding conflicts (detected as inconsistent updates of the knowledge locations).

The MAPE-K loop of the main concern *MAPE – K*(*FLEX_M*) (i.e., the flexibility concern in the role MASTER) can be therefore defined as follows:

$$MAPE - K(FLEX\_M) = \left\langle R_{FLEX\_M}, \overset{FLEX\_M}{\longrightarrow}, K(FLEX\_M) \right\rangle$$

where

$R_{FLEX\_M} = R_{CD} \bigcup R_{TMS} \bigcup R_{TS} = \{r_{M_{CD}}, r_{MA_{TMS}}, r_{PE_{TMS}}, r_{MA_{TS}}, r_{PE_{TS}}\}$
$\overset{FLEX\_M}{\longrightarrow} = \{(r_{M_{CD}}, r_{MA_{TMS}}),(r_{M_{CD}}, r_{MA_{TS}}),(r_{MA_{TS}}, r_{MA_{TMS}}),(r_{MA_{TMS}}, r_{PE_{TMS}}),(r_{MA_{TS}}, r_{PE_{TS}})\}$
$K(FLEX\_M) = K(CD) \bigcup K(TMS) \bigcup K(TS) =$ {congested,s_offer,m_offer,state,slaves,newSlave,change_master,getMaster}

Note that (according to Equation 3) the monitor computation of *MAPE – K*(*FLEX_M*) is performed by the rules $r_{M_{CD}}$, $r_{MA_{TMS}}$, and $r_{MA_{TS}}$ of R_{FLEX_M}. Rules $r_{M_{CD}}$ and $r_{MA_{TMS}}$ are in (intracomponent) interaction relation by means of the knowledge location *congested*; rules $r_{M_{CD}}$ and $r_{MA_{TS}}$ are in (intracomponent) interaction relation by means of locations *congested* and *s_offer*; $r_{MA_{TS}}$ and $r_{MA_{TMS}}$ are in (intracomponent) interaction relation by means of location *m_offer*; $r_{MA_{TMS}}$ directly interacts with rule $r_{PE_{TMS}}$; and $r_{MA_{TS}}$ directly interacts with rule $r_{PE_{TS}}$.

The monitor components of the instances of the loop *MAPE – K*(*FLEX_M*) on different cameras realize the information sharing pattern: the information shared among the cameras' organization controllers is represented by the signals *s_offer* and *m_offer* (knowledge locations for interaction of the master/slave election mechanism).

In Section 7, a qualitative comparison is made with the work presented in Iftikhar and Weyns [2012], where Timed Automata are used to model the same Traffic Monitoring case study and verify properties. In Arcaini et al. [2015], a comparison is made with the same work for the verification of properties related to the application requirements.

5. VALIDATION OF ADAPTATION REQUIREMENTS

Model validation is a model analysis activity and consists in investigating and/or executing a model to ensure that it reflects the user needs and statements about the application. In the context of modeling SA, besides testing the conformance between model and requirements, validation is useful to get a preliminary feedback of the correct operation of a MAPE-K control loop. Guarantee of such correctness can be given by formal verification of the interaction relation among rules involved in the control loop, but it requires more effort. Validation is, instead, less demanding than property verification and, therefore, applicable since the earlier stages of model development to detect faults and ambiguities with limited effort.

In our work of modeling self-adaptive systems, we have experimented with two model validation approaches, namely, *simulation* and *scenario-based validation*. Simulation is performed with the ASM simulator `AsmetaS` [Gargantini et al. 2008] and scenario-based validation with the validator `AsmetaV` [Carioni et al. 2008], both provided by the ASMETA framework [Arcaini et al. 2011]. During the validation process, the user

```
scenario smartHome1

load smartHome.asm

set roomTemp := 15;
set badAir := false;
set timePassed := false;
step
check time=EARLY_MORN and sgnHeaterFAIRLY_HOT_CH
and sgnWaterHeaterON_MWH;

step
check time=EARLY_MORN and setHeaterStatus=FAIRLY_HOT
and setWaterHeaterStatus=WE_ON;
```

```
step
check time=EARLY_MORN and heaterStatus=FAIRLY_HOT
                and waterHeaterStatus=WE_ON;

set roomTemp := 20;
set badAir := true;
set timePassed := true;
step
check time=MORN_AFT and sgnOpenWindow_AQ;

step
check time=MORN_AFT and setWindowStatus=OPEN;

step
check time=MORN_AFT and windowStatus=OPEN;
```

Code 6. Smart home—Example of scenario.

can directly simulate an ASM-based specification in an interactive way, or write a scenario that automatizes the simulation of the model and that checks that the produced outputs are as expected (see Section 5.1). We can apply these validation techniques several times during the development process. In this way, we are able to get enough confidence that the model we are developing is correctly modeling MAPE-K control loops as specified in the adaptation requirements.

In the following section, we present scenario-based validation of the smart home example (see Section 3.3). We refer to Arcaini et al. [2015] for a description of interactive simulation and scenario-based validation for the traffic monitoring case study.

5.1. Scenario-Based Validation

Scenario-based validation is an advanced way to simulate and inspect ASMs, by specifying a *scenario* representing an interaction sequence of actions of an external actor and activities of the machine as reaction to the actor actions. Scenarios are described in an algorithmic way using the textual language `Avalla` [Carioni et al. 2008] that provides constructs to **set** the environment (i.e., the values of monitored/shared functions), to **check** the machine state, to ask for the **exec**ution of certain transition rules, and to force the machine to make one **step** (or a sequence of steps by **step until**). The tool `AsmetaV` reads scenarios written in `Avalla` and executes them using the simulator `AsmetaS`; during simulation, `AsmetaV` captures any check violation and, if none occurs, it finishes with a *PASS* verdict.

Scenarios are a suitable way to reproduce and inspect the execution of MAPE-K loops. The first computation of a MAPE-K loop is context/self-aware monitoring, in which environment and managed system are monitored and, if some conditions hold, an analyze computation is triggered. Therefore, in order to reproduce a particular MAPE-K loop, by the **set** command we give to the system specific environment inputs, so that the desired control loop can start. Then, by the command **step**, we force a simulation step in which a MAPE computation is possibly executed. After a step of simulation is executed, we use the **check** command to verify that the knowledge has been updated correctly and that the subsequent steps (if any) of the MAPE-K loop have been triggered correctly with regard to the particular pattern instantiated by the loop under consideration. In addition, we can use the command **exec** to bring the system in a specific configuration required to trigger monitoring.

Code 6 shows an example of scenario for the smart home case study. The scenario simulates the situation in which control loops $MAPE - K(CH)$, $MAPE - K(MWH)$, and $MAPE - K(AQ)$ are executed. At the beginning, the monitoring phase of the loop related to the adaptation concern *CH* (comfortable heating) signals that adaptation is needed

when the sensor of room temperature detects 15°: in the first step, a signal to turn the heater on is sent (i.e., `sgnHeaterFAIRLY_HOT_CH`). Since it is early morning, also the loop *MAPE – K*(*MWH*) (related to hot water in the morning) is activated and a signal to turn the water heater on is sent (i.e., `sgnWaterHeaterON_MWH`). After another step, the signals have been received and the heater and the water heater are informed to turn themselves on (i.e., `setHeaterStatus=FAIRLY_HOT` and `setWaterHeaterStatus=WE_ON`) by the executors of the two loops. In the next step, the managed systems heater and water heater adapt accordingly. Afterwards, the loop *MAPE – K*(*AQ*) (regarding the air quality) is activated: when the sensor of air quality detects bad air, a signal to open the window is sent (i.e., `sgnOpenWindow_AQ`). After one step, the signal is received and the window is informed to open (i.e., `setWindowStatus=OPEN`) by the execute component of the loop. After one step, the managed window adapts accordingly.

6. VERIFICATION

Since validation techniques are not complete (i.e., they partially explore the state space of the specification), they cannot give us full assurance of the specification correctness. For this reason, on the final specification, we need to apply deeper analyses in terms of formal verification.

In our formal framework, we support verification approaches that deal with

(1) *system-independent properties* (or metaproperties), that is, properties that any self-adaptive model should guarantee;
(2) *MAPE-K correctness verification properties*, that is, properties to assess the correct interaction among MAPE computations in achieving an adaptation goal; and
(3) *requirement verification properties*, that is, properties representing adaptation goals related to the requirements of the specific system.

All approaches exploit the model checker `AsmetaSMV` [Arcaini et al. 2010a], a tool of the ASMETA framework that translates ASM specifications into models of the NuSMV model checker. The model checker allows the verification of *Computation Tree Logic* (CTL) and *Linear Temporal Logic* (LTL) formulas.

We here mainly consider the first two approaches, since we are principally interested in completely automatic techniques; however, we also give some small examples of the third approach. Section 6.1 presents the first approach, that is, model review in terms of metaproperty verification, while Section 6.2 describes the second approach, that is, the verification technique to check correctness of MAPE-K control loops by exploiting the interaction relation. Manual specification of requirement verification properties is shown in Section 6.3.

6.1. Model Review

This approach aims at determining if a model is of sufficient *quality* to be easy to develop, maintain, and enhance. This technique permits one to identify defects early in the system development, reducing the cost of fixing them. For this reason, it should be applied also on preliminary models. The `AsmetaMA` tool [Arcaini et al. 2010b] (based on `AsmetaSMV`) allows *automatic* review of ASMs. Typical vulnerabilities and defects that can be introduced during the modeling activity using ASMs are checked as violations of suitable *metaproperties* (*MP*, defined in Arcaini et al. [2010b] as CTL formulas). The violation of a metaproperty means that a quality attribute is not guaranteed, and it may indicate the presence of a real fault (i.e., the ASM is indeed faulty), or only of a *stylistic defect* (i.e., the ASM could be written in a better way).

In this work, we define some metaproperties tailored for self-adaptive systems:

—MP_{nc}: knowledge locations are *not in conflict*. Due to the distributed nature of the adaptation, MAPE computations (of the same control loop, or of different loops) might simultaneously update a same knowledge location to two different values. This definition corresponds to the definition of *inconsistent update* [Börger and Stärk 2003]. This fact might cause conflicting situations between control loops. In particular, we have a *single-loop* inconsistency if the inconsistent updates belong to the same MAPE-K loop (i.e., they are due to computations of the same control loop), and a *multiple-loop* inconsistency if the inconsistent updates belong to two different MAPE-K loops (i.e., they are due to computations of different control loops).
—MP_e: all rules involved in MAPE-K loops are *executed*. This metaproperty only guarantees that there is no over specification inside a MAPE-K loop formalization. However, it does not guarantee functional correctness of a MAPE-K loop. This can be checked by means of other verification techniques presented in Section 6.2.
—MP_m: the knowledge is *minimal*, that is, it does not contain locations that are *unnecessary* (they are never read or updated) or that do not assume all the values of their codomains. Note that a violation of this metaproperty may also indicate that the specification is not complete, that is, that the designer forgot to read/update a location.

6.1.1. Model Review of the Smart Home Case Study. As also reported in Song et al. [2013], some adaptation goals of the smart home case study are in conflict. These conflicts can be easily discovered by metaproperties MP_{nc}. The verification of these metaproperties over the model presented in Section 3.3 is as follows:

```
Location setWindowStatus is updated to values CLOSED and OPEN when are
satisfied simultaneously the conditions:
- (checkForAdaptation & execMAPE_MD & sgnCloseWindow_MD)
- (checkForAdaptation & execMAPE_AQ & sgnOpenWindow_AQ)
```

The metaproperty violation signals that loops $MAPE-K(MD)$ and $MAPE-K(AQ)$ are in conflict, as they can simultaneously ask to close and to open the window. In this case, the loop related to adaptation concern MD (i.e., minimize dispersion) detects that the window is open and the heater is turned on and, therefore, asks the agent managing the window to close it. At the same time, the loop related to adaptation concern AQ detects that the air quality is bad and, therefore, asks the agent managing the window to open it. Such conflict can be shown to the user who should decide which adaptation concern is more important, either the reduction of heating dispersion or the air quality.

As a further example of application of model review to SA systems, we refer the reader to Arcaini et al. [2015], where we discuss how the specification of the traffic monitoring case study was improved by applying model review that revealed a single-loop inconsistency and some minimality violations.

6.2. MAPE-K Correctness Verification

A MAPE-K control loop formalization is given by definition (2). In order to guarantee that the model correctly specifies the intended operation of a MAPE-K loop, it must be proved that the specification correctly captures the interaction relation $\xrightarrow{adj}$ on the set of rules R_{adj} associated to $MAPE-K(adj)$.

In the following, let us suppose that, for a given adaptation concern adj, rules r_{X_adj}, $r_{Y\_adj} \in R_{adj}$, and $r_{X\_adj} \xrightarrow{adj} r_{Y\_adj}$.

We can check the validity of the preceding interaction relation in a static and dynamic way. Note that we assume to be in the case of indirect interaction, since in case of direct interaction the relation is guaranteed by the structure of nested rules.

Verification of the Interaction Relation—Static Precondition. Statically, we can establish a necessary condition for yielding interaction between r_{X_adj} and r_{Y_adj}. Interaction is possible if rule r_{X_adj} updates locations of $K(adj)$ that are read in r_{Y_adj}. This can be checked by the predicate

$$inter(r_{X\_adj}, r_{Y\_adj}) \neq \varnothing, \tag{4}$$

where $inter(r_{X\_adj}, r_{Y\_adj})$ yields the set of function symbols of the *knowledge* vocabulary that are updated by r_{X_adj} and read by r_{Y_adj}.

Verification of the Interaction Relation—Dynamic Verification. Predicate (4) can be checked by static analysis on the structure of the ASM model. However, it only provides a necessary condition for making the interaction effective, that is, that there is some information written by r_{X_adj} that is read by r_{y_adj}; the property does not guarantee that the information is written and read in the correct order.

In order to check that the semantics of the interaction is respected, we must be able to specify properties regarding the execution of the rules. To this purpose, we have to give some definitions.

Definition 6.1 (*Rule Firing Condition*). Given an ASM model M, we define *rule firing condition* the function

$$RFC : Rules(M) \rightarrow Conditions(M),$$

where *Rules* is the set of transition rules of M and *Conditions* are Boolean predicates over the state of M. *RFC* associates to each rule the condition that must be satisfied in order for the rule to be executed.

The technique to statically compute *RFC* of a rule r of an ASM M is described in Arcaini et al. [2010b]: it builds the conjunction of all conditions that precede r in the control flow graph of M. Note that any kind of ASM is supported, from single agent to sync/async multiagent ASMs. We here give a concrete example on how it is calculated.

Example 6.2 (*Smart Home*). In Code 2, the *RFC* of the update rule `sgnHeaterFAIRLY_HOT_CH := true` is `checkForAdaptation and monMAPE_CH and not(roomTemp < 10) ∧ roomTemp < 18`.

Definition 6.3 (*Effectiveness*). A rule is *effective* if it produces a nonempty update set, that is, if at least one of its update rules fires. Effectiveness of a rule r is defined as follows:

$$eff(r) = \bigvee_{ur \in UpRules(r)} RFC(ur) \tag{5}$$

being $UpRules(r)$ the set of update rules nested in r.

In terms of self-adaptive systems, a rule modeling a computation is effective if it produces some information that can be later used by other computations.

Definition 6.4 (*Established interaction*). The execution of a rule r_1 establishes an interaction with a rule r_2 if a function contained in $inter(r_1, r_2)$ is updated. We define the *established interaction* as follows:

$$estInt(r_1, r_2) = \bigvee_{ur \in UpRules(r_1, inter(r_1, r_2))} RFC(ur) \tag{6}$$

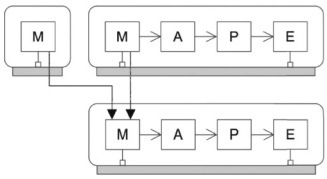

Fig. 6. Instance of a MAPE pattern.

being $UpRules(r_1, inter(r_1, r_2))$ the set of update rules nested in r_1 that update functions in $inter(r_1, r_2)$.

In terms of self-adaptive systems, an interaction is established between two rules modeling components X and Y if X sends some information to Y.

We now exploit the previous definitions to verify that the interaction relation between the two rules r_{X_adj} and r_{Y_adj} is guaranteed in the model. We have to prove the following LTL[7] property:

$$\mathbf{G}(eff(r_{Y\_adj}) \textbf{ implies } \mathbf{O}(estInt(r_{X\_adj}, r_{Y\_adj}))). \tag{7}$$

Property (7) assures that whenever rule modeling computation Y is effective, then it has read information computed in the past by rule modeling the X computation.

However, in case of adaptation control decentralized among multiple components, it may happen that more rules $r_{X_i\_adj} \in R_{adj}$, $i = 1, \ldots, n$ interact with the same computation r_{Y_adj} (i.e., $r_{X_i\_adj} \xrightarrow{adj} r_{Y\_adj}$). Consider, for example, the interaction of MAPE components established by the coordination model in Figure 6: the two M computations of the first row both interact with the M computation in the second row.

In this case, Property (7) is not adequate to check correctness of the interaction relation, which depends on the semantics given to the set of interactions from components X_i to component Y. Therefore, if the execution of rule r_{Y_adj} requires the previous firing of *all* rules $r_{X_i\_adj}$, then, to verify the interaction relation among rules $r_{X_i\_adj}$ and r_{Y_adj}, we have to prove the property:

$$\bigwedge_{i=1}^{n} \mathbf{G}(eff(r_{Y\_adj}) \textbf{ implies } \mathbf{O}(estInt(r_{X_i\_adj}, r_{Y\_adj}))). \tag{8}$$

Otherwise, in case the execution of *at least one* of the rules $r_{X_i\_adj}$ is required, we have to prove the property:

$$\bigvee_{i=1}^{n} \mathbf{G}(eff(r_{Y\_adj}) \textbf{ implies } \mathbf{O}(estInt(r_{X_i\_adj}, r_{Y\_adj}))). \tag{9}$$

6.2.1. Verification on the Case Studies. We here show how the static and the dynamic verification can be used to check that the MAPE-K control loops of the smart home and traffic monitoring case studies are implemented correctly.

Smart Home. The smart home case specification has five MAPE-K loops (see Sections 3.3 and 4.1); the first four loops have one indirect interaction each, while the fifth one has two indirect interactions. The static verification that the components of the five MAPE-K loops can communicate is therefore done with the following six formulas (see Equation (4)):

[7]**G** is the Always-LTL operator *globally* requiring that a property holds on the entire path, and **O** is the Past-LTL operator *once* requiring that something happened in the past.

inter(r_checkRoomTempMAPE_CH, r_adaptHeaterMAPE_CH) =
{sgnHeaterOFF_CH, sgnHeaterFAIRLY_HOT_CH, sgnHeaterVERY_HOT_CH} $\neq \varnothing$
inter(r_checkWindowAndHeaterMAPE_MD, r_adaptWindowMAPE_MD) = {sgnCloseWindow_MD} $\neq \varnothing$
inter(r_checkAirQualityMAPE_AQ, r_adaptWindowMAPE_AQ) = {sgnOpenWindow_AQ, sgnCloseWindow_AQ} $\neq \varnothing$
inter(r_checkHotWaterMorningMAPE_MWH, r_adaptWaterHeaterMAPE_MWH) = {sgnWaterHeaterON_MWH} $\neq \varnothing$
inter(r_checkElectrMAPE_ES, r_adaptHeaterMAPE_ES) = {sgnHeaterFAIRLY_HOT_ES} $\neq \varnothing$
inter(r_checkElectrMAPE_ES, r_adaptWaterHeaterMAPE_ES) = {sgnWaterHeaterOFF_ES} $\neq \varnothing$

For the first four MAPE-K loops, the verification is trivial since M rules simply write signals in the knowledge that are read by the E rules. For the fifth MAPE-K loop (*ES*), there are two different E rules (`r_adaptHeaterMAPE_ES` and `r_adaptWaterHeaterMAPE_ES`) that can be triggered by the M rule `r_checkElectrMAPE_ES`. Indeed, the monitoring rule, which detects that the heater is set to the maximum speed and the water heater is on, can nondeterministically choose either to decrease the speed of the heater or turn the water heater off.

The dynamic verification of the six indirect interactions is performed using the following six LTL properties (see Equation 7):

G(((checkForAdaptation and execMAPE_CH and sgnHeaterOFF_CH) or
(checkForAdaptation and execMAPE_CH and sgnHeaterFAIRLY_HOT_CH) or
(checkForAdaptation and execMAPE_CH and sgnHeaterVERY_HOT_CH)) implies
O((checkForAdaptation and monMAPE_CH and roomTemp < 10) or
(checkForAdaptation and monMAPE_CH and not(roomTemp < 10) and roomTemp < 18) or
(checkForAdaptation and monMAPE_CH and not(roomTemp < 10) and not(roomTemp < 18))))
G((checkForAdaptation and execMAPE_MD and sgnCloseWindow_MD) implies
O(checkForAdaptation and monMAPE_MD and heaterStatus != OFF and windowStatus = OPEN))
G(((checkForAdaptation and execMAPE_AQ and sgnOpenWindow_AQ) or
(checkForAdaptation and execMAPE_AQ and sgnCloseWindow_AQ)) implies
O((checkForAdaptation and monMAPE_AQ and badAir) or (checkForAdaptation and monMAPE_AQ and not(badAir))))
G((checkForAdaptation and execMAPE_MWH and sgnWaterHeaterON_MWH) implies
O(checkForAdaptation and monMAPE_MWH and time = EARLY_MORN))
G((checkForAdaptation and execMAPE_ES and sgnHeaterFAIRLY_HOT_ES) implies
O(checkForAdaptation and monMAPE_ES and time = MORN_AFT and heaterStatus = VERY_HOT and
waterHeaterStatus = WE_ON and esAdaptHeater))
G((checkForAdaptation and execMAPE_ES and sgnWaterHeaterOFF_ES) implies
O(checkForAdaptation and monMAPE_ES and time = MORN_AFT and heaterStatus = VERY_HOT and
waterHeaterStatus = WE_ON and not(esAdaptHeater)))

Traffic Monitoring. The traffic monitoring case study (see Section 4.1) contains an example of a complex MAPE pattern; the MAPE-K loop related to the flexibility concern implements the information sharing pattern in which three M components share information. In order to verify the correct implementation of the pattern (i.e., the interactions in $\overset{FLEX\_M}{\longrightarrow}$), we only need to prove the indirect interactions between M components, as the other interactions (i.e., interactions between M and A, P, and E components) are direct and, therefore, their correctness is guaranteed by the model structure.

There are three interactions between the M components: $(r_{M_{CD}}, r_{MA_{TMS}})$, $(r_{M_{CD}}, r_{MA_{TS}})$, and $(r_{MA_{TS}}, r_{MA_{TMS}})$. Their static verification is as follows:

inter$(r_{M_{CD}}, r_{MA_{TMS}})$ = {congested $\neq \varnothing$}
inter$(r_{M_{CD}}, r_{MA_{TS}})$ = {congested, s_offer} $\neq \varnothing$
inter$(r_{MA_{TS}}, r_{MA_{TMS}})$ = {m_offer} $\neq \varnothing$

In a multiagent ASM as the traffic monitoring specification, a rule can be executed by different agents (of the same type); we identify with `ag.r` an *instance* of rule `r`, that is, the execution of rule `r` by agent `ag`. In the case study, for each camera, there is an organization controller $\texttt{org}_i$ $(i = 1, \ldots, n)$ that executes rules $r_{M_{CD}}$, $r_{MA_{TMS}}$, and $r_{MA_{TMS}}$.

Dynamic verification can only reason in terms of concrete executions and, therefore, requires the instantiation of the abstract interactions using instantiated rules.[8] In the case study, for each organization controller $\texttt{org}_i$, the three abstract interactions are instantiated as follows: $(\texttt{org}_i.r_{M_{CD}}, \texttt{org}_i.r_{MA_{TMS}})$, $(\texttt{org}_i.r_{M_{CD}}, \texttt{org}_{i+1}.r_{MA_{TS}})$, $(\texttt{org}_{i+1}.r_{MA_{TS}}, \texttt{org}_i.r_{MA_{TMS}})$. Now, for each organization controller $\texttt{org}_i$, Equation (7) can be instantiated for the three interactions as follows.

G((stateOC(org$_i$) = MASTER and congested(org$_i$) and m_offer(cameraOC(org$_i$))) implies
  O(stateOC(org$_i$) = MASTER and not(stopCam(cameraOC(org$_i$))) and cong(cameraOC(org$_i$)) and not(congested(org$_i$))))
G((stateOC(org$_{i+1}$) = MASTER and congested(org$_{i+1}$) and not(m_offer(cameraOC(org$_{i+1}$))) and s_offer(cameraOC(org$_{i+1}$)))) implies
  O(stateOC(org$_i$) = MASTER and not(stopCam(cameraOC(org$_i$))) and cong(cameraOC(org$_i$)) and not congested(org$_i$)))
G((stateOC(org$_i$) = MASTER and congested(org$_i$) and m_offer(cameraOC(org$_i$))) implies
  O(stateOC(org$_{i+1}$) = MASTER and congested(org$_{i+1}$) and not(m_offer(cameraOC(org$_{i+1}$))) and s_offer(cameraOC(org$_{i+1}$))))

Note that the verifications of the first and the third formulas are an instantiation of Equation (8): indeed, rule $\texttt{org}_i.r_{MA_{TMS}}$ requires information from both rules $\texttt{org}_i.r_{M_{CD}}$ and $\texttt{org}_{i+1}.r_{MA_{TS}}$ (as depicted in Figure 6).

6.3. Requirement Verification Properties

This approach regards properties related to the application requirements, such as invariants, general properties on the controlled part of the system, and adaptation goals expressed as reachable and liveness conditions, that can be verified as classical temporal properties. An extensive example of this approach can be found in Arcaini et al. [2015], where a number of CTL properties were proved to guarantee correctness and reliability of the traffic monitoring case study (similarly to what was done in Iftikhar and Weyns [2012]). Other examples of ASM model checking exist in literature [Arcaini et al. 2016b].

We here show some representative properties for the smart home case study.

6.3.1. Requirement Verification Properties of the Smart Home Case Study. Liveness properties specify that the system can eventually reach desired states. In our case study, we check that the heater, the window, and the water heater can always modify their statuses to all their possible values.

```
(forall $s in HeaterStatus with ag(ef(heaterStatus =$s)))
(forall $s in WindowStatus with ag(ef(windowStatus =$s)))
(forall $s in WaterHeaterStatus with ag(ef(waterHeaterStatus =$s)))
```

Reachability properties permit one to check that particular configurations of the system can be reached. For example, we can check that there exists a state in which the window is open and the heater is turned on.

```
ef(windowStatus = OPEN and heaterStatus != OFF)
```

Note that this state triggers the execution of $MAPE-K(MD)$ that tries to minimize the heating dispersion. Indeed, MAPE-K correctness verification (see Section 6.2) checks that a MAPE-K loop is executed correctly, not that it can be executed. Therefore, if we are interested in knowing whether some particular adaptations are eventually performed, we have to specify suitable reachability properties.

Safety properties specify that the system remains in *safe* states. However, self-adaptive systems allow the system to reach unsafe states (those that may trigger the adaptation); from an unsafe state, a suitable MAPE-K loop could be designed to bring the system back to a safe state. For this case study, we do not specify particular safety properties, since states we consider unsafe are those that trigger the five

[8]This information cannot be automatically derived from the model but must be given by the modeler.

Table I. Traffic Monitoring Case Study—Experimental Results

	Size	Model review		MAPE-K correctness verification	
# of cameras	(# of BDD nodes)	# of properties	time (s)	# of properties	Time (s)
4	64,326	10,816	9,493	9	9
5	96,563	38,875	TIMEOUT	12	29
6	95,353	111,672	TIMEOUT	15	51
7	217,590	N/A	N/A	18	133
8	389,777	N/A	N/A	21	250
9	526,650	N/A	N/A	24	994
10	12,172,189	N/A	N/A	27	1,161

adaptations previously described. Of course, also in self-adaptive systems there could be (particularly unsafe) situations that should never occur, and their absence should be checked by means of appropriate safety properties.

6.4. Scalability

All the experiments have been executed on a Linux PC with an Intel(R) Core(TM) i7-5600U CPU (2.60GHz), and 8GB of RAM. Table I shows the size (in terms of the number of BDD nodes[9] of the NuSMV translation of the ASM model) of the specification of the traffic monitoring case study for an increasing number of cameras. Moreover, it also shows, for the model review and MAPE-K verification techniques, the number of properties that must be checked and the execution time. As expected, the execution time grows with the number of cameras. However, while for MAPE-K verification we are able to verify also the setting with 10 cameras in less than 20 minutes, the model review technique timeouts (we set a timeout of 3 hours) already with five cameras.[10] For both techniques, the addition of a camera makes the state space larger. However, in the MAPE-K verification each added camera requires one to check only three additional properties: therefore, the additional computational burden is still feasible. For the model review, instead, each additional camera requires one to check roughly the triple of the properties, and this is the reason why the technique does not scale; for example, for the case with four cameras the model review produces 10,816 properties, and for the case with five cameras the number of properties grows to 38,875. As future work, we plan to devise abstraction techniques for model review in order to be able to handle big specifications.

7. COMPARISON WITH THE STATE OF THE ART

SA has been widely studied in the software architecture community [Allen et al. 1998]. Various mechanisms and frameworks for handling adaptation have been proposed, such as SA with aspect orientation, Dynamic Reconfiguration, Model-Driven Development frameworks for SA, and frameworks for self-optimization (including the adaptation cost itself) [Cheng et al. 2009; de Lemos et al. 2013; Kephart and Chess 2003; Morin et al. 2009; Cardellini et al. 2012; Mirandola et al. 2014a]. However, as shown in Weyns et al. [2012], little attention has been given in the past to formal modeling and analysis of self-adaptive systems. In particular, as revealed in the Dagstuhl seminar on *Software Engineering for Self-Adaptive Systems: Assurances* [de Lemos et al. 2014], one emerging key challenge is *requirements assurance* that consists in providing evidence that a self-adaptive system satisfies its functional and nonfunctional requirements during operation.

[9]The model has been executed using the NuSMV option `dynamic` that executes a dynamic reordering of the variables and, in this way, usually permits one to reduce the number of BDD nodes.
[10]For more than six cameras, we are not even able to produce all the properties to check and, therefore, to perform the verification.

Here we review the main approaches that use formal methods for the design and analysis of SA. More precisely, we identify generally agreed *key features* that a formal approach should have for modeling and analyzing distributed self-adaptive systems, and we use them as criteria for comparing our work with existing ones, also in relation to some open issues in the field. Then, we provide a clear position about the scope and applicability of our approach with regard to these desired features.[11]

7.1. Support for Effective Modeling and Analysis Techniques

We here review approaches for modeling, validating, and verifying functional requirements of self-adaptive systems, since these are our primary aim.

Automata-based or transition-based computational models have been advocated for adaptation, such as the S[B] systems [Merelli et al. 2012] and Synchronous Adaptive Systems (SAS) of MARS [Adler et al. 2007]. They share a multilevel view of SA. They rely on a multilayered model reminiscent of hierarchical state machines and automata. In the simple case of two layers, the lower behavioral level describes the actual dynamic behavior of the system and the upper structural level accounts for the dynamically changing environmental constraints imposed on the lower system. Petri Nets extensions also exist for dealing with adaptation. The work in Zhang and Cheng [2006], for example, combines Petri Nets modeling with LTL for property checking, including correctness of adaptations and robustness properties of adaptive programs.

In the area of concurrency, classical Process Algebras (CCS, CSP, ACP) have been tailored, such as in Bartels and Kleine [2011], to the modeling of self-adaptive systems as a subclass of reactive systems. The approach SOTA (State Of The Affairs) [Abeywickrama and Zambonelli 2012] supports an early, goal-level, model checking analysis for adaptive systems. However, they adopt a very complex model checking process involving several formalisms: the i* framework is used for modeling static aspects, an operational SOTA language is defined and used to describe the dynamic aspects and dependencies among components, and process calculus Finite State Processes (FSP) and asynchronous first-order linear-time temporal logic (FLTL) code of the model checker Labeled Transition System Analyzer (LTSA) are then provided to formally define the goal or utility for verification purposes. In addition to specific temporal properties specified for a particular model, the framework can also check general properties that any model should assure (e.g., absence of deadlock). In Djoudi et al. [2014], a formal model for context-aware adaptive systems is proposed by establishing a three-layered separation among system components, context entities, and management components. Relationships between layers are dynamically established via the generation of strategies by the management layer. Maude is adopted as a semantic framework for the proposed model, and Maude reflection and metaprogramming capabilities are exploited to enrich it with context-awareness concepts. Formal analysis is done using the Maude model checker. Güdemann et al. [2007] presents a case study of an adaptive production automation cell modeled in Lustre—a typed synchronous dataflow language with a discrete time model—using the SCADE Suite and the verification of functional properties.

We also considered approaches relying on state- and machine-based formalisms similar to ASMs such as the B method, Alloy, and Z. The authors in Lanoix et al. [2011] present an approach to the formal specification and verification of dynamic reconfigurations of component-based systems using the B method for the specification of component architectures and FTPL–a logic based on architectural constraints and

[11]Please note that certain criteria may be cross-cutting and overlapping to some degree.

on event properties, translatable into LTL—to express temporal properties over reconfiguration sequences to be model checked. Georgiadis et al. [2002] uses architectural constraints specified in Alloy for the specification, design, and implementation of self-adaptive architectures for distributed systems. Magee and Maibaum [2006] outlines an approach for modeling and analyzing fault tolerance and self-adaptive mechanisms in distributed systems. The authors use a modal action logic formalism, augmented with deontic operators, to describe normal and abnormal behavior.

All the formalization approaches mentioned previously do not support the explicit modeling of feedback loops for SA and their properties. The actual feedback control loops are hidden or abstracted. In our approach, instead, we clearly explore MAPE-K feedback loops as a means to identify and enact adaptation, so elevating them to first-class entities in the ASM formal specification of a self-adaptive system. Moreover, most of these formal approaches to SA assume a centralized point of control.

The authors in Bruni et al. [2013] present an essential model of *Adaptable Transition Systems*. The same authors propose in Bruni et al. [2015] a conceptual framework for adaptation centered on the role of control data and its realization in a reflective logical language like Maude by using the Reflective Russian Dolls model. They exploit the statistical model checker PVeStA and present robot swarms equipped with obstacle-avoidance self-assembly strategies as case study. The proposed computational model for SA is, however, built around hierarchical structures of managing layers. To really capture the distributed nature of self-adaptive systems, more coordination patterns of managing components/agents need to be employed [Weyns et al. 2013].

According to a decentralized feedback loop-based approach, a general goal-oriented modeling framework [Abeywickrama et al. 2013], called SOTA and tool-supported by SimSOTA (an Eclipse plug-in), is being developed to support modeling, simulation, and validation of self-adaptive systems. Similarly to our approach, SOTA aims at supporting the development of self-adaptive systems by allowing one to validate correctness of decentralized feedback loop models. However, unlike our formal approach, SOTA adopts a semiformal notation, namely, UML activity diagrams, as primary notation to model the behavior of feedback loops.

In Iftikhar and Weyns [2012], Timed Automata are used to model the Traffic Monitoring case study and verify properties. A network of Timed Automata allows specifying the behavior of MAPE components that synchronize through clock variables and interact via channels. The Uppaal model checker is used to verify flexibility and robustness properties expressed in Timed Computation Tree Logic (TCTL)—a computational tree logic extended with clock variables. The same case study is specified using the Z method in Weyns et al. [2010a], and is revised in Vromant et al. [2011] as an example of multiple subloops within a single control loop and interacting control loops. In Gil de la Iglesia and Weyns [2013] the same formal approach is used to realize the self-adaptive layer of a Mobile-Learning Application and TCTL is used to specify and verify four groups of properties: functional correctness, GPS service adaptation, self-healing, and MAPE-K loops interference. Recently, Gil De La Iglesia and Weyns [2015] introduced formal model templates of the behaviors of MAPE-based feedback loops in terms of networks of Timed Automata, and property specification templates that support verification of the correctness of the adaptation behaviors by Uppaal. The primary focus of this template-based approach is on the design and verification of a "specific family" of self-adaptive systems, namely, a target domain of distributed applications in which self-adaptation is used for managing resources for robustness and openness requirements via adding and removing resources from the system. These works are (as far as we know) the main efforts in presenting a formal approach to specify and verify behavioral properties of decentralized self-adaptive systems through MAPE-K feedback loops, and this is why we were mainly inspired by them.

From a formal specification point of view, however, we find formalisms as Timed Automata and Z less flexible and intuitive than ASMs: they have rigid notations, and require specific skills to develop and understand models that are often not very concise. Differently, ASMs allow a direct and natural formalization of computing concepts through programming practice and mathematical standards. Therefore, practitioners can work with ASMs without any further explanation, viewing them as "pseudocode over abstract data," which comes with a well-defined semantics. Moreover, refinement and model composition allow the application of the method to the specification of large-scale self-adaptive systems. In addition, flexibility and abstractness of our framework allow us to model adaptation strategies and coordination schemes of control loops, and to discover and analyze conflicts that may arise when MAPE-K subloops deal with subconcerns of the main adaptation concern.

Regarding the modeling style, our framework does not impose any restriction on the kind of models that can be written. A modeler must only annotate rules involved in a MAPE-k loop with appropriate annotations, that are then used by the verification framework to check the loop correctness. Other approaches, instead, provide more restrictive ways of specifying MAPE components in terms of *templates* [Gil De La Iglesia and Weyns 2015].

Regarding property verification, our rigorous specifications permit requirements verification by model checking but we do not support time constraints. For example, for the Traffic Monitoring case study, we have been able to prove in Arcaini et al. [2015] the same properties as in Iftikhar and Weyns [2012] since time is not involved. However, we contribute with additional validation and verification strategies to check the model against MAPE components interactions and control loop interferences. In addition, by exploiting the concept of "interaction relation," we provide a formal ASM semantics to a graphical notation used to express instances of MAPE patterns.

7.2. Support for Feedback Control Loops and Decentralized SA

Feedback loops are cornerstones of self-adaptive systems. On the one hand, assurances improve the realization of feedback loops in self-adaptive systems; on the other hand, feedback loops contribute to provide assurances about the controlled system since, for example, they facilitate the identification of the core phenomena to control and the realization of composable analysis tasks that can be applied incrementally along the adaptation loop [de Lemos et al. 2014]. Therefore, feedback loops should be modeled and analyzed explicitly like in our formal framework. In some previous formalization approaches (see Section 7.1), instead, feedback loops are hidden or abstracted. Another important feature, which our framework supports, is the capability to specify decentralized adaptation control in terms of feedback loops. In real distributed self-adaptive systems the control is completely decentralized; that is, there is no central authority and the system's organization itself may fluctuate dynamically as a result of self-adaptive units that join and leave the system.

Other contributions exist that explicitly model MAPE-k feedback loops. To name a few: Bruni et al. [2015], Weyns et al. [2013], Abeywickrama et al. [2013], Vromant et al. [2011], and Gil De La Iglesia and Weyns [2015].

7.3. Support for Design Patterns

Support for design patterns is a systematic engineering approach for applying the principles of software architecture to the organization of self-adaptive systems. The work in Weyns et al. [2013] presents architectural design patterns of interacting MAPE components. Such patterns specify components interaction from a structural viewpoint. Our framework supports the modeling of general behavioral patterns of interactive

MAPE-K feedback loops as those identified in Weyns et al. [2013], and provides computational semantics to the graphical notation used to specify pattern instances. The framework also supports verification strategy useful to guarantee that certain interaction patterns among components are actually captured in the system model.

7.4. Support for Modeling and Analyzing Timed Adaptation

Specification and verification of self-adaptive systems are very difficult to carry out when involving time constraints. The functional correctness of the system and of its adaptation logic depends, in fact, also on the time associated with events. Most of the existing techniques are not effective when dealing with real-time constraints, because quantitative temporal aspects are not taken into account. Some existing formal methods (such as Timed Automata and Petri Nets) have been extended for the modeling of time and timed adaptation (see Musliner [2000], Zeller and Prehofer [2012], Prehofer and Zeller [2012], Iftikhar and Weyns [2014], and Camilli et al. [2015], to name a few). They mainly employ model-checking techniques for verifying time constraints. The need for adaptation models targeted for real-time systems that treat the duration and overhead of adaptation as first class entities is, however, still challenging.

Our ASM-based framework does not provide any notion of time and, therefore, timed adaptation is not allowed.

7.5. Support at Runtime

Some approaches use formal methods at design time, and others at runtime or a combination of both [Calinescu and Kikuchi 2011]. Revising existing runtime verification techniques or developing new ones from analysis techniques that work mainly at design time (e.g., model checking) is still challenging. The survey in Calinescu and Kikuchi [2011] and the Dagstuhl seminar [de Lemos et al. 2014] describe challenges and existing approaches to employ formal methods at runtime, that is, to achieve runtime adaptation using mathematically based techniques from the area of formal methods. Among these, the approach ActivFORMS (Active FORmal Models for Self-adaptation) [Iftikhar and Weyns 2014] is a preliminary result. Its aim is to guarantee that the adaptation goals verified offline (i.e., at design time) are guaranteed also at runtime. It adopts an integrated formal model of a MAPE-K loop (i.e., models of the knowledge and the adaptation components) that is directly executed by a virtual machine at runtime and can be dynamically changed with changing goals.

In Calinescu et al. [2015], a Runtime Quantitative Verification (RQV) approach, called DECIDE, is proposed to develop decentralized self-adaptive distributed systems that continue to meet their QoS requirements after failures and environment changes. DECIDE is applicable to systems that exhibit stochastic behavior, and involves Continuous-Time Markov Chains (CTMCs), which are used to describe the behavior of the system's components formally, and the temporal logic Continuous Stochastic Logic (CSL), which is used to express the properties of CTMCs.

Currently, we do not deal with runtime concerns. Our research focuses on providing requirements assurance for self-adaptive systems during system design. As future work, we plan to integrate our ASM-based framework with runtime monitoring techniques that the ASM formalism already supports [Arcaini et al. 2012].

7.6. Support for Uncertainty

Due to unpredictable changes, for example in the environment, a self-adaptive system may have no control over new unexpected processes that influence the environment and the system's organization itself that (especially in a decentralized setting) may fluctuate dynamically. In spite of recent advances on the adoption of formal methods

for dealing with uncertainty [Esfahani et al. 2011; Esfahani and Malek 2013; Perez-Palacin and Mirandola 2014], managing uncertainty in self-adaptive systems is still an impervious engineering problem. Artificial self-organizing systems have shown to be particularly robust to dynamic operating conditions [Weyns et al. 2010b]. Formal approaches based on stochastic behaviors for making decisions under probability theory are also promising, but they are known to be computationally expensive for execution, which makes them unsuitable for use at runtime, where often decisions have to be made very fast [Esfahani and Malek 2013].

Dealing with uncertainty is not supported. Although definitions of probabilistic ASMs have been proposed [Beierle and Kern-Isberner 2003], they are not mature enough to be applied to SA.

7.7. Support for Advanced Self-Adaptive Systems

AI mechanisms and biological metaphors are increasingly gaining consideration [Satoh 2012; Fernandez-Marquez et al. 2013] as ways for realizing more degrees of autonomicity and, therefore, more reliable self-adaptive systems. Modeling and analysis techniques for these advanced autonomic systems are missing, although agent technologies have been identified as a key enabler for their engineering.

The ASM framework already supports a general notion of "agent" for capturing distributed computation. We intend in the future to extend such a notion taking inspiration from AI and biological systems. In particular, we foresee in the capacity of an ASM agent to change (adapt) its own program at runtime a basic building block for realizing more complex autonomicity and different types of policies (action, goal, and utility policies) in autonomic computing [Kephart and Walsh 2004].

8. EVALUATION OF THE APPROACH

Self-adaptive systems are generally difficult to specify, validate, and verify due to their high complexity and dynamic nature. Particularly, when involving decentralized adaptation, the system adaptive behavior is the result of the collaborative behavior of multiple managing agents and components responsible for enabling adaptation.

In this section, we evaluate our formalism in terms of advantages offered for modeling SA and of its shortcomings.

8.1. Advantages

Modeling self-adaptation features was possible thanks to the multiagent computational model available in ASMs to specify distributed computation and coordination among agents. Other characteristics of the ASMs, such as model compositionality, rules combination by means of powerful rule constructors, and mechanisms for agents' interaction (shared functions), helped us to specify agents' coordination and communication, also in the presence of decentralized control among multiple MAPE components and in the presence of interacting subloops.

We achieve *clear separation of concerns*: (i) between adaptation logic and functional logic since we model managing and managed components as separated agents; (ii) among MAPE computations that are modeled by separated transition rules; (iii) among different adaptation concerns that are specified by distinguished sets of rules and interaction relations; (iv) between decentralized and centralized loop's control; between direct and indirect components interaction. This separation of concerns helps the designer to focus on one adaptation activity at a time, and for each adaptation aspect, separate the adapting parts from the adapted ones. This helps in keeping the system complexity under control, and also facilities reasoning about components behavior due to model conciseness.

The availability of a set of tools for model analysis helped us in different activities:

—*Validate adaptation requirements*. By executing the specification or constructing scenarios, reproducing precise system configurations, we get feedback of the operation of the MAPE-K control loops.
—*Determine conflicting MAPE-K loops*. By checking for inconsistent knowledge updates, we can discover conflicts between simultaneous executions of different adaptive behaviors. However, a deeper model analysis, by means of the proof of suitable metaproperties, can reveal single/multiple-loop inconsistencies inside the agents' programs. Such conflicting situations often require one to reason about *priorities* of adaptation concerns, which can be established by scheduling the agents' operations.
—*Check for MAPE-K correctness*. The formalism is able to express semantics of the MAPE components interaction by means of the interaction relation. We support a verification strategy, based on static and dynamic analysis, useful to assess correctness of the components interaction.
—*Check for conformance to MAPE pattern*. The interaction relation also allows one to express the operational semantics of MAPE patterns; therefore, the strategy for checking MAPE correctness is also used for checking the execution of instances of MAPE patterns.
—*Assert the system correctness*. Model checking can be used to verify properties, expressed as temporal logic formulas, related to the requirements of the application.

In this work, we have only considered the specification of self-adaptive systems and not their implementation that, however, could be linked to the high-level model by exploiting the refinement mechanism of the ASMs. *Model refinement* is one of the constituents of the ASM method: it allows one to build a system in an incremental way, through a chain of refined models until a level where the specification can be easily mapped to the final implementation. At each refinement step, a refined model must be proved to be a *correct refinement* [Börger and Stärk 2003] of an abstract one, namely, it must be guaranteed that each run of the refined machine can be matched with a run of the abstract machine (using a given conformance relation between states). A tool exists for checking refinement correctness [Arcaini et al. 2016a] in an automatic way. In the chain of refined models, the implementation is usually seen as the last step of refinement; it could be obtained by automatic translation of the ASM, or it could be developed independently. In the latter case, the conformance of the implementation with regard to the specification must be checked. We can check the conformance in two ways: at development time using model-based testing [Arcaini et al. 2014], or at runtime using runtime monitoring [Arcaini et al. 2012].

8.2. Shortcomings

The approach has some shortcomings. Some have already been discussed in Section 7 with regard to the state of the art. We here review the main ones.

Some limits of the approach are due to ASMs, adopted as an underlying formal method of our framework. For example, dealing with uncertainty and time adaptation is not supported by our framework as ASMs do not have any notion of stochastic behaviors and of time. Considering these aspects would require a theoretical extension of ASMs themselves. Moreover, ASMs are not appropriate for handling nonfunctional requirements (quality properties). ASMs are suitable for describing the behavior of systems and, therefore, the primary goal of our framework is the modeling and analysis of functional requirements of adaptive systems, and providing a formal operation semantics to MAPE patterns. In many frameworks for SA existing in the literature, little attention is given to the formalization of the adaptation components themselves, which is important to provide guarantees of correctness of the adaptation [Iftikhar

and Weyns 2014]. Moreover, a precise way to express the computational meaning of components interaction (graphically rendered by the use of arrows) in MAPE patterns is missing.

Some other limits, instead, are intrinsic of the proposed framework. Properties used for model review and for the verification of MAPE-K correctness (see Sections 6.1 and 6.2) are automatically derived from the ASM model and then translated, together with the model, to NuSMV for verification. Such translation introduces an overhead that limits the scalability of the verification approach (in addition to the classical state explosion problem of model checking). However, as seen in Section 6.4, the verification of MAPE-K correctness scales better than model review; indeed, while model review is very general and produces several properties to check different qualities of the specification (minimality, consistency), verification of MAPE-K correctness is more specific and produces few properties to check that the MAPE pattern has been executed correctly.

9. CONCLUSION AND FUTURE DIRECTIONS

In this article, we introduced the concept of self-adaptive ASMs, based on the notion of multiagent ASMs, as a formal framework to specify self-adaptive systems. The framework permits one to write an ASM model of a self-adaptive system and provides a technique to identify the components involved in a MAPE-k loop and how these components interact. The framework permits one to describe both simple MAPE-k loops or more complex loops structured according to some patterns. Thanks to the formalization of interaction between components of a MAPE-k loop, some temporal properties can be automatically derived from the specification to check that the components involved in a MAPE-k loop communicate correctly and are executed in the correct order (as, for example, specified by the MAPE pattern). In addition to this novel technique, the framework can also exploit classical validation and verification techniques for ASMs, as scenario-based validation and model review (here tailored for self-adaptive ASMs).

The proposed framework does not provide a direct support for modeling uncertainty, timing aspects, and quality properties. However, the integration of the framework with others designed to address these missing aspects is feasible. As future work, we plan to consider the approach in Mirandola et al. [2014b] to deal with nonfunctional properties for SA, and to exploit appropriate extensions of ASMs with time models [Graf and Prinz 2007] for specifying time-triggered adaptation. We also plan to consider runtime monitoring techniques [Arcaini et al. 2012] to connect our formal model to a runtime adaptation middleware and study the conformance relation between the model and the real system execution at runtime.

REFERENCES

Dhaminda B. Abeywickrama, Nicklas Hoch, and Franco Zambonelli. 2013. SimSOTA: Engineering and simulating feedback loops for self-adaptive systems. In *Proceedings of the International C* Conference on Computer Science and Software Engineering (C3S2E'13)*. ACM, New York, NY, 67–76.

Dhaminda B. Abeywickrama and Franco Zambonelli. 2012. Model checking goal-oriented requirements for self-adaptive systems. In *Proceedings of the IEEE 19th International Conference and Workshops on Engineering of Computer-Based Systems (ECBS'12)*. IEEE, 33–42.

Rasmus Adler, Ina Schaefer, Tobias Schuele, and Eric Vecchié. 2007. *From Model-Based Design to Formal Verification of Adaptive Embedded Systems*. Springer, Berlin, 76–95.

Robert Allen, Rémi Douence, and David Garlan. 1998. Specifying and analyzing dynamic software architectures. In *Proceedings of the 1st International Conference on Fundamental Approaches to Software Engineering (FASE'98), held as part of the Joint European Conferences on Theory and Practice of Software (ETAPS'98)*, Egidio Astesiano (Ed.). Springer, Berlin, 21–37.

Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. 2010a. AsmetaSMV: A way to link high-level ASM models to low-level NuSMV specifications. In *Abstract State Machines, Alloy, B and Z*, Lecture Notes in Computer Science, Vol. 5977, Marc Frappier, Uwe Glässer, Sarfraz Khurshid, Régine Laleau, and Steve Reeves (Eds.). Springer, Berlin, 61–74.

Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. 2010b. Automatic review of abstract state machines by meta property verification. In *Proceedings of the 2nd NASA Formal Methods Symposium (NFM'10), NASA/CP-2010-216215*, César Muñoz (Ed.). NASA, Langley Research Center, Hampton VA, 4–13.

Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. 2012. CoMA: Conformance monitoring of java programs by abstract state machines. In *Runtime Verification*, Lecture Notes in Computer Science, Vol. 7186, Sarfraz Khurshid and Koushik Sen (Eds.). Springer, Berlin, 223–238.

Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. 2014. Offline model-based testing and runtime monitoring of the sensor voting module. In *ABZ 2014: The Landing Gear Case Study*, Frédéric Boniol, Virginie Wiels, Yamine Ait Ameur, and Klaus-Dieter Schewe (Eds.). Communications in Computer and Information Science, Vol. 433. Springer International Publishing, 95–109.

Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. 2016a. SMT-based automatic proof of ASM model refinement. In *Proceedings of the 14th International Conference on Software Engineering and Formal Methods, held as part of STAF 2016*, Rocco De Nicola and Eva Kühn (Eds.). Springer International Publishing, Cham, 253–269.

Paolo Arcaini, Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. 2011. A model-driven process for engineering a toolset for a formal method. *Softw.: Pract. Exper.* 41, 2 (2011), 155–166.

Paolo Arcaini, Roxana-Maria Holom, and Elvinia Riccobene. 2016b. ASM-based formal design of an adaptivity component for a cloud system. *Form. Aspect. Comput.* 28, 4 (2016), 567–595.

Paolo Arcaini, Elvinia Riccobene, and Patrizia Scandurra. 2015. Modeling and analyzing MAPE-K feedback loops for self-adaptation. In *Proceedings of the 10th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'15)*. IEEE, 13–23.

Björn Bartels and Moritz Kleine. 2011. A CSP-based framework for the specification, verification, and implementation of adaptive systems. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'11)*. ACM, New York, NY, 158–167.

Christoph Beierle and Gabriele Kern-Isberner. 2003. Modelling conditional knowledge discovery and belief revision by abstract state machines. In *Abstract State Machines 2003: Proceedings of the 10th International Workshop on Advances in Theory and Practice (ASM'03)*, Egon Börger, Angelo Gargantini, and Elvinia Riccobene (Eds.). Springer, Berlin, 186–203.

Egon Börger and Robert Stärk. 2003. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag.

Yuriy Brun, Giovanna Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi Müller, Mauro Pezzè, and Mary Shaw. 2009. Software engineering for self-adaptive systems. Springer-Verlag, Berlin, pp. 48–70.

Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch Lafuente, and Andrea Vandin. 2013. *Adaptable Transition Systems*. Springer, Berlin, 95–110.

Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch Lafuente, and Andrea Vandin. 2015. Modelling and analyzing adaptive self-assembly strategies with Maude. *Sci. Comput. Program.* 99, C (March 2015), 75–94.

Radu Calinescu, Simos Gerasimou, and Alec Banks. 2015. *Proceedings of the 18th International Conference on Fundamental Approaches to Software Engineering (FASE'15), held as part of the European Joint Conferences on Theory and Practice of Software (ETAPS'15)*. Springer, Berlin, pp. 235–251.

Radu Calinescu and Shinji Kikuchi. 2011. *Foundations of Computer Software. Modeling, Development, and Verification of Adaptive Systems: 16th Monterey Workshop 2010, Revised Selected Papers*. Springer, Berlin, pp. 122–135.

Matteo Camilli, Angelo Gargantini, and Patrizia Scandurra. 2015. Specifying and verifying real-time self-adaptive systems. In *Proceedings of the 26th IEEE International Symposium on Software Reliability Engineering (ISSRE'15)*. IEEE, 303–313.

Valeria Cardellini, Emiliano Casalicchio, Vincenzo Grassi, Stefano Iannucci, Francesco Lo Presti, and Raffaela Mirandola. 2012. MOSES: A framework for QoS driven runtime adaptation of service-oriented systems. *IEEE Trans. Softw. Eng.* 38, 5 (Sept. 2012), 1138–1159.

Alessandro Carioni, Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. 2008. A scenario-based validation language for ASMs. In *Abstract State Machines, B and Z*, Lecture Notes in Computer Science, Vol. 5238, Egon Börger, Michael Butler, Jonathan P. Bowen, and Paul Boca (Eds.). Springer, Berlin, 71–84.

Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Di Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. 2009. *Software Engineering for Self-Adaptive Systems: A Research Roadmap*. Lecture Notes in Computer Science, Vol. 5525. Springer, Berlin. pp. 1–26.

Rogerio de Lemos, David Garlan, Carlo Ghezzi, and Holger Giese. 2014. Software engineering for self-adaptive systems: Assurances (Dagstuhl seminar 13511). *Dagstuhl Rep.* 3, 12 (2014), 67–96. http://drops.dagstuhl.de/opus/volltexte/2014/4508

Rogério de Lemos, Holger Giese, Hausi A. Müller, Mary Shaw, Jesper Andersson, Marin Litoiu, Bradley Schmerl, Gabriel Tamura, Norha M. Villegas, Thomas Vogel, Danny Weyns, Luciano Baresi, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Ron Desmarais, Schahram Dustdar, Gregor Engels, Kurt Geihs, Karl M. Göschka, Alessandra Gorla, Vincenzo Grassi, Paola Inverardi, Gabor Karsai, Jeff Kramer, Antónia Lopes, Jeff Magee, Sam Malek, Serge Mankovskii, Raffaela Mirandola, John Mylopoulos, Oscar Nierstrasz, Mauro Pezzè, Christian Prehofer, Wilhelm Schäfer, Rick Schlichting, Dennis B. Smith, João Pedro Sousa, Ladan Tahvildari, Kenny Wong, and Jochen Wuttke. 2013. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*. Lecture Notes in Computer Science, Vol. 7475. Springer, Berlin, 1–32.

Brahim Djoudi, Chafia Bouanaka, and Nadia Zeghib. 2014. Model checking pervasive context-aware systems. In *Proceedings of the 2014 IEEE 23rd International WETICE Conference (WETICE'14)*. IEEE Computer Society, Washington, DC, 92–97.

Naeem Esfahani, Ehsan Kouroshfar, and Sam Malek. 2011. Taming uncertainty in self-adaptive software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE'11)*. ACM, New York, NY, 234–244.

Naeem Esfahani and Sam Malek. 2013. *Uncertainty in Self-Adaptive Software Systems*. Springer, Berlin, 214–238.

Jose Luis Fernandez-Marquez, Giovanna Di Marzo Serugendo, Sara Montagna, Mirko Viroli, and Josep Lluis Arcos. 2013. Description and composition of bio-inspired design patterns: A complete overview. *Nat. Comput.* 12, 1 (2013), 43–67.

Héctor Garcia-Molina. 1982. Elections in a distributed computing system. *IEEE Trans. Comput.* 31, 1 (Jan. 1982), 48–59.

Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. 2008. A metamodel-based language and a simulation engine for abstract state machines. *J. UCS* 14, 12 (2008), 1949–1983.

Ioannis Georgiadis, Jeff Magee, and Jeff Kramer. 2002. Self-organising software architectures for distributed systems. In *Proceedings of the 1st Workshop on Self-Healing Systems (WOSS'02)*. ACM, New York, NY, 33–38.

Didac Gil de la Iglesia and Danny Weyns. 2013. Guaranteeing robustness in a mobile learning application using formally verified MAPE loops. In *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'13)*. IEEE Press, 83–92.

Didac Gil De La Iglesia and Danny Weyns. 2015. MAPE-K formal templates to rigorously design behaviors for self-adaptive systems. *ACM Trans. Auton. Adapt. Syst.* 10, 3, Article 15 (Sept. 2015), 31 pages.

Susanne Graf and Andreas Prinz. 2007. Time in state machines. *Fundam. Inf.* 77, 1–2 (Jan. 2007), 143–174.

Matthias Güdemann, Andreas Angerer, Frank Ortmeier, and Wolfgang Reif. 2007. Modeling of self-adaptive systems with SCADE. In *International Symposium on Circuits and Systems (ISCAS'07)*. IEEE, 2922–2925.

Markus C. Huebscher and Julie A. McCann. 2008. A survey of autonomic computing—Degrees, models, and applications. *ACM Comput. Surv.* 40, 3, Article 7 (Aug. 2008), 28 pages.

M. Usman Iftikhar and Danny Weyns. 2012. A case study on formal verification of self-adaptive behaviors in a decentralized system. In *Proceedings of the 11th International Workshop on Foundations of Coordination Languages and Self Adaptation (FOCLASA'12), EPTCS*, Vol. 91. 45–62.

M. Usman Iftikhar and Danny Weyns. 2014. ActivFORMS: Active formal models for self-adaptation. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'14)*. ACM, New York, NY, 125–134.

Jeffrey O. Kephart and David M. Chess. 2003. The vision of autonomic computing. *Computer* 36, 1 (2003), 41–50.

Jeffrey O. Kephart and William E. Walsh. 2004. An artificial intelligence perspective on autonomic computing policies. In *Proceedings of the 5th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'04)*. IEEE Computer Society, 3–12.

Arnaud Lanoix, Julien Dormoy, and Olga Kouchnarenko. 2011. Combining proof and model-checking to validate reconfigurable architectures. *Electron. Notes Theor. Comput. Sci.* 279, 2 (Dec. 2011), 43–57.

Jeff Magee and Tom Maibaum. 2006. Towards specification, modelling and analysis of fault tolerance in self managed systems. In *Proceedings of the 2006 International Workshop on Self-Adaptation and Self-Managing Systems (SEAMS'06)*. ACM, New York, NY, 30–36.

Emanuela Merelli, Nicola Paoletti, and Luca Tesei. 2012. A multi-level model for self-adaptive systems. In *Proceedings of the 11th International Workshop on Foundations of Coordination Languages and Self Adaptation (FOCLASA'12)*, *EPTCS*, Vol. 91. 112–126.

Raffaela Mirandola, Pasqualina Potena, Elvinia Riccobene, and Patrizia Scandurra. 2014b. A reliability model for service component architectures. *J. Syst. Softw.* 89, C (March 2014), 109–127.

Raffaela Mirandola, Pasqualina Potena, and Patrizia Scandurra. 2014a. Adaptation space exploration for service-oriented applications. *Sci. Comput Program.* 80, Part B, 0 (2014), 356–384.

Brice Morin, Olivier Barais, Gregory Nain, and Jean-Marc Jezequel. 2009. Taming dynamically adaptive systems using models and aspects. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*. IEEE Computer Society, Washington, DC, 122–132.

David J. Musliner. 2000. Imposing real-time constraints on self-adaptive controller synthesis. In *Proceedings of the 1st International Workshop on Self-Adaptive Software (IWSAS'00)*. Springer-Verlag, New York, 143–160.

George A. Papadopoulos and Farhad Arbab. 1998. *Coordination Models and Languages*. Technical Report. Centre for Mathematics and Computer Science, Amsterdam, The Netherlands.

Diego Perez-Palacin and Raffaela Mirandola. 2014. Uncertainties in the modeling of self-adaptive systems: A taxonomy and an example of availability evaluation. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE'14)*. ACM, New York, NY, 3–14.

Christian Prehofer and Marc Zeller. 2012. Towards runtime adaptation in real-time, networked embedded systems. In *Proceedings of the 7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*. IEEE, 271–274.

Ichiro Satoh. 2012. In *Proceedings of the 9th International Conference on Distributed Computing and Artificial Intelligence*. Springer, Berlin, pp. 221–228.

Hui Song, Stephen Barrett, Aidan Clarke, and Siobhán Clarke. 2013. In *Proceedings of the 16th International Conference on Model-Driven Engineering Languages and Systems (MODELS'13)*. Springer, Berlin, pp. 555–571.

Pieter Vromant, Danny Weyns, Sam Malek, and Jesper Andersson. 2011. On interacting control loops in self-adaptive systems. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'11)*. ACM, New York, NY, 202–207.

Danny Weyns, M. Usman Iftikhar, Didac Gil de la Iglesia, and Tanvir Ahmad. 2012. A survey of formal methods in self-adaptive systems. In *Proceedings of the 5th International C* Conference on Computer Science and Software Engineering (C3S2E'12)*. ACM, New York, NY, 67–79.

Danny Weyns, Sam Malek, and Jesper Andersson. 2010a. FORMS: A formal reference model for self-adaptation. In *Proceedings of the 7th International Conference on Autonomic Computing (ICAC'10)*. ACM, New York, NY, 205–214.

Danny Weyns, Sam Malek, and Jesper Andersson. 2010b. On decentralized self-adaptation: Lessons from the trenches and challenges for the future. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'10)*. ACM, New York, NY, 84–93.

Danny Weyns, Bradley R. Schmerl, Vincenzo Grassi, Sam Malek, Raffaela Mirandola, Christian Prehofer, Jochen Wuttke, Jesper Andersson, Holger Giese, and Karl M. Göschka. 2013. *Software Engineering for Self-Adaptive Systems II: International Seminar, 2010 Revised Selected and Invited Papers*. Springer, Berlin, pp. 76–107.

Marc Zeller and Christian Prehofer. 2012. Timing constraints for runtime adaptation in real-time, networked embedded systems. In *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'12)*. IEEE Press, 73–82.

Ji Zhang and Betty H. C. Cheng. 2006. Model-based development of dynamically adaptive software. In *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*. ACM, New York, NY, 371–380.

Received October 2015; revised August 2016; accepted November 2016