Modeling and Analyzing MAPE-K Feedback Loops for Self-adaptation

Paolo Arcaini
DIGIP
Università degli Studi di Bergamo, Italy
Email: paolo.arcaini@unibg.it

Elvinia Riccobene
Dipartimento di Informatica
Università degli Studi di Milano, Italy
Email: elvinia.riccobene@unimi.it

Patrizia Scandurra
DIGIP
Università degli Studi di Bergamo, Italy
Email: patrizia.scandurra@unibg.it

***Abstract*—The MAPE-K (Monitor-Analyze-Plan-Execute over a shared Knowledge) feedback loop is the most influential reference control model for autonomic and self-adaptive systems. This paper presents a conceptual and methodological framework for formal modeling, validating, and verifying distributed self-adaptive systems. We show how MAPE-K loops for self-adaptation can be naturally specified in an abstract stateful language like Abstract State Machines. In particular, we exploit the concept of multi-agent Abstract State Machines to specify decentralized adaptation control by using MAPE computations. We support techniques for validating and verifying adaptation scenarios, and getting feedback of the correctness of the adaptation logic as implemented by the MAPE-K loops. In particular, a verification technique based on meta-properties is proposed to allow discovering unwanted interferences between MAPE-K loops at the early stages of the system design. As a proof-of-concepts, we model and analyze a traffic monitoring system.**

I. Introduction

Modern software systems typically operate in dynamic environments and deal with highly changing operational conditions: components can appear and disappear, may become temporarily or permanently unavailable, may change their behavior, etc. Self-Adaptation (SA) has been widely recognized [17], [18], [26] as an effective approach to deal with the increasing complexity, uncertainty and dynamicity of these systems. A well recognized engineering approach to realize self-adaptation is by means of a feedback control loop called *MAPE-K* [26], [13] and conceived as a sequence of four computations *Monitor-Analyze-Plan-Execute* over a *Knowledge* base.

To provide guarantee of the functional correctness of the adaptation logic, formal methods can be used as a rigorous means for specifying and reasoning about self-adaptive systems' behavior, both at design time and at runtime. However, the survey in [34] shows that, although the attention for self-adaptive software systems is gradually increasing, the number of studies that employ formal methods remains low, and mainly related to runtime verification. Formally founded design models that cover both structural and behavioral aspects of self-adaptation and of approaches to validate and verify behavioral properties are highly demanded. Of extreme importance is engineering self-adaptive systems in a way that unwanted interferences/conflicts between feedback control loops are excluded at the early stages of the system design.

This work is part of our ongoing research activity [31], [32] on formal modeling self-adaptive systems having a *decentralized MAPE-K control loop* architecture. As in [36], with decentralization, we refer to how control decisions in a self-adaptive software system are coordinated among different components, regardless of how those control components are physically distributed. Specifically, we consider decentralization at the level of the four computations of a MAPE-K loop.

By assuming a clear *separation of adaptation concerns* and exploiting the theoretical framework of the *multi-agent Abstract State Machines* (ASM) [12], we here show how to model and analyze in ASM the behavior of a self-adaptive system by representing MAPE-K loops explicitly and naturally formalized in terms of agents' actions (ASM transition rules). Most existing formal approaches to SA assume a centralized point of control and none of them, except [24], [35], [1] (see Sect. II), provides a semantic description of the adaptation logic from the perspective of feedback control loops.

Our framework supports also formal techniques for validating and verifying adaptation scenarios, and getting feedback (already at system design time) of the correctness of the adaptation logic as implemented by the MAPE-K loops. In addition to common state-of-the-art model checking techniques for verifying properties of self-adaptive systems, we support the analysis of systems where multiple MAPE-K loops interact (and possibly operate on top of each other) by detecting possible intra- and inter- loop interferences and conflicts. To this purpose, a verification technique based on the proof of meta-properties is proposed to allow discovering unwanted interferences between MAPE-K loops.

The rest of this paper is organized as follows. Sect. II presents related work. Sect. III describes the reference model we adopt for realizing SA, and Sect. IV introduces the Traffic Monitoring System here taken as running case study. Sect. V provides some background on the ASM formal method. Sect. VI presents the proposed ASM-based modeling and analysis framework for self-adaptive systems. The ASM model of the Traffic Monitoring System is shown in Sect. VII. In Sect. VIII we present the results of applying the model validation and verification techniques on the case study by, respectively, simulating adaptation scenarios and verifying meta-properties and properties through model checking for system adaptation concerns such as flexibility and robustness.

Sect. IX discusses some faced challenges. Sect. X concludes the paper and outlines future directions of our work.

II. Related Work

Our proposal takes inspiration by many former works on formal modeling and analysis of self-adaptive systems.

Automata-based or transition-based computational models have been advocated for adaptation, such as the S[B] systems [30] and Synchronous Adaptive Systems (SAS) of MARS [3]. They have in common a multi-level view of SA. They rely on a multi-layered 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 [37], 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 Algebra (CCS, CSP, ACP) have been tailored, such as in [10], to the modeling of self-adaptive systems as a subclass of reactive systems. The approach SOTA [2] 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 formal model checker Labeled Transition System Analyzer (LTSA) is 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 [19] 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 meta-programming capabilities are exploited to enrich it with context-awareness concepts. Formal analysis is performed using the Maude model checker. [23] presents a case study of an adaptive production automation cell modeled in the Lustre language – a typed synchronous dataflow language with a discrete time model – using the SCADE Suite and the verification of functional properties.

We also considered approached relying on state- and machine-based formalisms close to ASMs such as the B method, Alloy and Z. The authors in [28] present an approach to the formal specification and verification of dynamic re-configurations of component-based systems using the B method for the specification of component architectures and FTPL – a logic based on architectural constraints and on event properties, translatable into LTL – to express temporal properties over (re-)configuration sequences to model-check. [21] uses architectural constraints specified in Alloy for the specification, design and implementation of self-adaptive architectures for distributed systems. [29] 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.

As important as their contribution could be to support the specification and analysis of self-adaptive systems, the formalization approaches mentioned above 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.

In [24], Timed Automata are used to model the Traffic Monitoring case study (the same presented here) and verify (with the tool Uppaal) flexibility and robustness properties by model checking. The same case study is specified using the Z method in [35]. These are (to the best of our knowledge) the first works presenting a formal approach to specify and verify behavioral properties of decentralized self-adaptive systems through MAPE-K feedback loops, and this is the reason why we mainly inspired to them. However, the approaches in [24], [35] suffer from the over-specification due to the rigidity of the formalisms Timed Automata and Z. Differently, the natural expression in ASM of fundamental computing concepts through programming practice and mathematical standards allow the practitioner to work with ASMs without any further explanation, viewing them as "pseudocode over abstract data" which comes with a well defined semantics supporting the intuitive understanding [12]. Moreover, the simplicity and modularity of the resulting ASM formal specification allow the application of the method to the specification of large-scale self-adaptive systems.

The Traffic Monitoring System has been revised in [33] where the simple sequence of four MAPE computations to form a control loop is extended in case of camera failures to (i) allow the execution of multiple sub-loops within a single control loop – *intra-loop coordination*, and (ii) enable MAPE computations across loops to coordinate the various phases of adaptation – *inter-loop coordination*. The flexibility and abstractness of our framework allow us to model any adaptation strategy and coordination schema of control loops, and to discover and analyze conflicts that may arise also in complex scenarios where MAPE-K sub-loops deal with sub-concerns of the main adaptation concern.

The authors in [14] present an essential model of *Adaptable Transition Systems*. The same authors in [15] propose a conceptual framework for adaptation centered around the role of control data and its realization in a reflective logical language

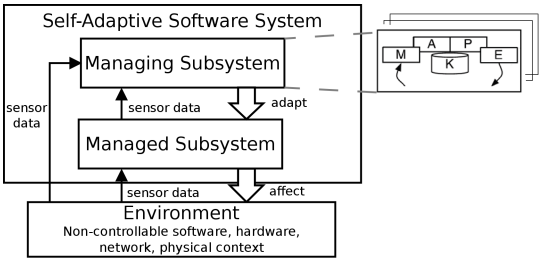

Fig. 1: A self-adaptive software system (adapted from [36])

like Maude by using the Reflective Russian Dolls model. They also exploit the statistical model checker PVeStA and present a 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 [36].

According to a decentralized feedback loop-based approach, a general goal-oriented modeling framework [1], called SOTA (State Of The Affairs) and tool-supported by SimSOTA (an Eclipse plug-in), is being developed to support the modeling, simulation and validation of self-adaptive systems. Similarly to our approach, SOTA aims at supporting the development of self-adaptive systems by allowing to validate the actual correctness of decentralized feedback loop models before implementation. However, unlike our formal approach, SOTA adopts a semi-formal notation, namely UML activity diagrams, as primary notation to model the behavior of feedback loops.

A recently proposed approach is ActivFORMS (Active FORmal Models for Self-adaptation) [25]. 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. We postpone as future work to study the conformance relation between the formal model and the real system execution at runtime.

III. Reference Model for Self-Adaptation

According to the conceptual model FORMS [35][1] and the study in [36], SA is based on the design principle of *separation of concerns*. As shown in Fig. 1 (adapted from [36]), a self-adaptive system is situated in an environment (both physical and software entities) and basically consists of a two-layer architecture: a *managed subsystem* layer that comprises the application logic, and a *managing subsystem* on top of the managed subsystem comprising the adaptation logic. This last realizes a *feedback loop* that monitors the environment and the managed subsystem, and adapts the latter when necessary,

[1]FORMS is a metamodel consisting of formally specified modeling elements that correspond to the key concerns in the design of self-adaptive software systems, and a set of relationships that guide their composition.

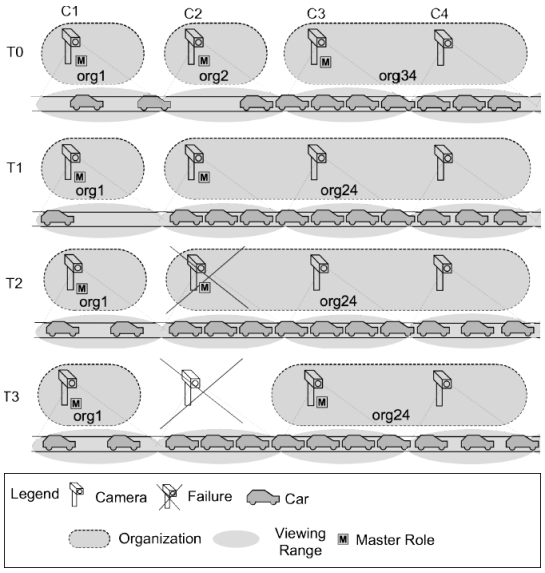

Fig. 2: Adaptation scenarios (adapted from [24])

such as to deal with particular types of faults (self-heal), self-optimize when operating conditions change, self-reconfigure when a goal changes, etc. Typically, the managing subsystem is conceived as a set of interacting feedback loops, one per each self-adaptation aspect (or concern). Other layers can be added to the system where higher-level managing subsystems manage underlying subsystems, which can be managing systems themselves.

A common approach to realize a feedback loop is by means of a MAPE-K (Monitor-Analyze-Plan-Execute over a Knowledge base) [26] 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 particular data from the underlying managed system and the environment through *probes* (or *sensors*) of the managed system, and saves data in the Knowledge. A component Analyze (A) performs data analysis to check whether an adaptation is required. If so, it triggers a component Plan (P) that composes a workflow of adaptation actions necessary to achieve the system's goals. These actions are then carried out by a component Execution (E) through *effectors* (or *actuators*) of the managed system.

Computations M, A, P, and E may be made by multiple components that coordinate with one another to adapt the system when needed, i.e., they may be decentralized throughout the multiple MAPE-K loops [36]. These MAPE components can communicate explicitly or indirectly by sharing information in the knowledge repository.

IV. Running Case Study

As case study to exemplify hereinafter our formal framework for self-adaptive systems, we present a traffic monitoring application inspired by the work in [24].

A number of intelligent cameras along a road (see Fig. 2) are endowed with a data processing unit and a communication

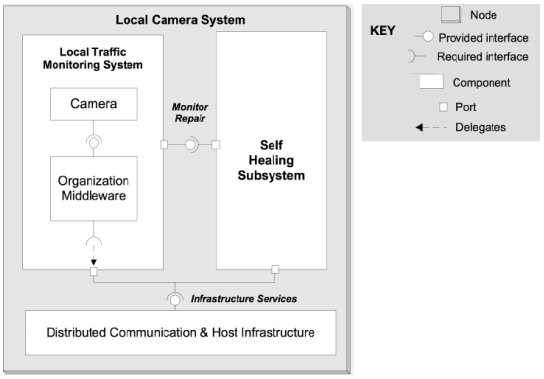

Fig. 3: Camera system architecture (adapted from [24])

unit to interact with each other. Cameras have to collaborate to observe larger phenomena such as traffic jams and aggregate the monitored data. Traffic jams can span the viewing range of multiple cameras and can dynamically grow and dissolve. Cameras enter or leave the collaboration whenever the traffic jam enters or leaves their viewing range.

Fig. 3 shows the architecture of a camera system. Every local camera (the component *Camera*) provides the functionality to detect traffic jams and inform clients. To access the hardware and communication facilities on the camera, the local camera system can rely on the services provided by the distributed communication and host infrastructure. Camera collaboration dynamics are managed by an organization controller (the component *Organization Middleware*). The organization controller is responsible for restructuring camera organizations and for adapting the system. The need for self-adaptation is triggered by changes in the surrounding environment, namely the congestion detected by a sensor of the camera – a subcomponent *Traffic Monitor* of the camera component –, and by the failures of other cameras (silent nodes) as detected by the component *Self Healing Subsystem* using a *Heartbeat* interaction pattern. Organization controllers of cameras collaborate to manage an organization that spans multiple cameras. To simplify synchronization issues, a *master/slave* control model is used at intra-organizational level. For each organization, one of the organization controllers is elected as master, whereas the other controllers of the organization are slaves. The master is responsible for managing the dynamics of that organization by synchronizing with all of the slaves.

According to the reference model for SA presented in Sect. III, the managed system is the local camera, while the managing subsystem corresponds to the self-healing subsystem and the organization middleware. These last components implement MAPE-K control loops for two main adaptation concerns[2]. The first concern is system *flexibility* for the dynamic adaptation of an organization. See, for example, the scenario in Fig. 2 from configuration T0 to T1, where camera 2 joins the organization of cameras 3 and 4 after it monitors a traffic jam. The second concern is adaptation to external camera failures (*robustness*), i.e., when a failing camera becomes unresponsive without sending any incorrect data. This scenario is shown in Fig. 2 from T2 to T3, where camera 2 fails. A further MAPE-K loop is also used to deal with internal failures of the camera (*robustness*).

[2]The vision of MAPE-K loops we take here slightly differ from the one taken in [24].

V. Background on ASMs

ASMs are an extension of FSMs [12] where unstructured control states are replaced by states comprising arbitrary complex data, and transitions are expressed by rules describing how data change from one state to the next. ASM *states* are multi-sorted first-order structures, i.e., domains of objects with functions and predicates (boolean functions) defined on them. State function values are saved into *locations* which may be updated from one state to another by firing a set of *transition rules* (or machine program).

The basic form of a transition rule is the *guarded update* "**if** *Condition* **then** *Updates*", where function *Updates* – having form $f(t_1, \ldots, t_n) := t$ – are simultaneously executed when *Condition* – a first order formula – is true. Besides the guarded rule, there is a finite set of *rule constructors* to model simultaneous parallel actions (`par`), sequential actions (`seq`), non-determinism (`choose`), unrestricted synchronous parallelism (`forall`), domain extension (`extend`). Due to their parallel execution, we require updates to be consistent, i.e., no pair of updates exist at a time which try to 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), *controlled* (read and written by the machine), *shared* (read/written by the machine/environment).

ASMs allow to model any kind of computational paradigm, from a *single* agent executing simultaneous parallel actions, to distributed *multiple* agents interacting in a synchronous or asynchronous way. A *multi-agent ASM* is given by a family of pairs $(a, ASM(a))$, where each agent $a : Agent$ has a "local" view, $View(a, S)$, of the *global state* S (see Fig. 4). An agent executes its own (possibly the same but differently instantiated) machine $ASM(a)$ specifying its local behavior, and contributes to determine the next state S'. A predefined function *program* on *Agent* indicates the ASM associated with an agent, and it is used to dynamically associate behavior to agents. Within transition rules, each agent can identify itself by means of a special 0-ary function $self : Agent$ which is interpreted by each agent a as itself.

For a complete theoretical definition of the ASM multi-agent model, we refer the reader to [12].

The ASM formal method is supported by the tool-set ASMETA (ASM mETAmodeling) [7], [8] for model editing, validation and verification.

VI. Self-adaptive ASMs

Here we present a formal framework, based on the ASM formalism, to model a self-adaptive system according to a

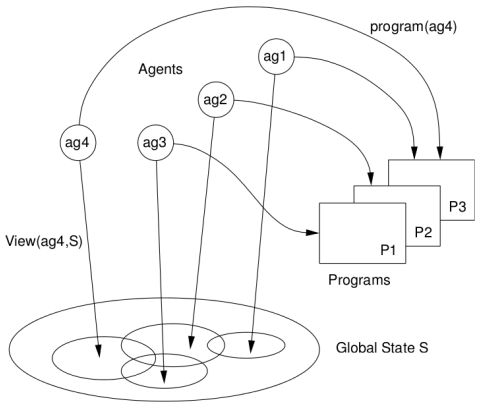

Fig. 4: Global state and partial view in a multi-agent ASM (taken from [12])

system's layered architecture with decentralized MAPE-K control loops.

Because of the distributed nature of self-adaptive systems, we use the notion of *multi-agent ASMs*. A software component is therefore represented in terms of an ASM agent able to interact with other agents (components). All together these ASM agents comprise the logic of a distributed ASM, called *self-adaptive ASM*, that provides the overall functionality of the self-adaptive system. It is, therefore, able to monitor the environment and itself and to self-adapt accordingly.

According to Sect. III, the self-adaptive layer (the managing subsystem) is conceived as a set of interacting MAPE-K loops. Therefore, the set *Agent* of a self-adaptive ASM is the union of the set MgA of *managing agents* and the set MdA of *managed agents*. Managing agents encapsulate the logic of self-adaptation of the MAPE-K loops, while managed agents encapsulate the system's functional logic.

A self-adaptive system may expose a certain number of MAPE-K loops, $\{\mathit{MAPE}(adj_1), \ldots, \mathit{MAPE}(adj_n)\}$, one per each adaptation concern adj_i. The behavior of a $\mathit{MAPE}(adj_i)$ is conceptually captured by a set of ASM transition rules:

$$\mathit{MAPE}(adj_i) = \{R^{a_1}_{\mathit{MAPE}(adj_i)}, \ldots, R^{a_m}_{\mathit{MAPE}(adj_i)}\}$$

where $\{a_1 \ldots, a_m\} \subseteq MgA$ are the managing agents involved in the execution of the loop $\mathit{MAPE}(adj_i)$, and $R^{a_j}_{\mathit{MAPE}(adj_i)}$ is the ASM rule modeling the behavior of the agent a_j, $j = 1, \ldots, m$, in the loop $\mathit{MAPE}(adj_i)$.

Since in a decentralized setting a single managing agent $a_j \in MgA$ may be involved in one or more loops $\mathit{MAPE}(adj_i)$, $i = 1, \ldots, n$, its program takes the form:

$$\mathit{program}(a_j) = \textbf{par} R^{a_j}_{\mathit{MAPE}(adj_{j_1})}, \ldots, R^{a_j}_{\mathit{MAPE}(adj_{j_k})} \textbf{endpar}$$

and consists in the parallel execution of its behavioral contributions to the $j_1, \ldots, j_k$ loops the agent is involved in. These rules are annotated (as comments //) with appropriate labels @M_c (for context-aware monitoring), @M_s (for self-aware monitoring), @A (for analyzing), @P (for planning), and @E (for execution)[3], depending on the role of the agent a in the loop.

An adaptation concern is also characterized by a *knowledge* $K(adj)$. In ASM, this is an ASM module defining only signature (domains and functions symbols) shared among the managing agents of the MAPE-K loop $\mathit{MAPE}(adj)$ to maintain representations of the managed subsystem (the *reflective model* [35]) and the environment, and other useful information for the enactment of the MAPE computations. The union of all these knowledges leads to an unique common knowledge $K = \bigcup_{adj} K(adj)$ shared by all managing agents.

The notion of *environment* is directly supported in the ASM theory by means of ASM monitored functions (see Sect. V). *Probes* (or *sensors*) from the environment are therefore modeled as ASM monitored functions or derived functions (the result of an elaboration of monitored functions). Probes from the managed system are instead represented by the changing value of ASM controlled functions modeling internal information to aware the system about itself. *Actuators* (or *effectors*) are specified in terms of actions (ASM rules) of managed agents that update controlled locations according to the adaptation plan decided by managing agents.

In the subsections below, we describe how to express in ASM the key mechanisms of the reference model FORMS [35] for realizing SA through MAPE-K feedback loops.

A. Context- and Self- Awareness

According to [35], an *update computation* perceives the state of the environment, while a *monitor computation* perceives the state of the managed subsystem to update the system model (the observed data from the system) in the knowledge. Update computations and monitor computations may trigger analyze computations when particular conditions hold. An *analyze computation* assesses the collected data from the environment and/or the system itself to determine the system's ability to satisfy its goals. Update computations in combination with analyze computations provides for context-awareness [35], which is a key property of self-adaptive systems. Monitor and analyze computations provide for self-awareness [35], which is another key property.

We use the terms *context-aware monitoring* and *self-aware monitoring* to denote update computations and monitor computations, respectively. In managing components, these computations are explicitly captured by ASM rules annotated with @M_c and @M_s, respectively. Context-aware monitoring consists into looking values of ASM monitored locations (the environment), while self-aware monitoring consists into looking values of ASM controlled locations representing internal locations of the managed system.

A rule scheme for a context/self-aware monitoring computation of a MAPE-K loop may take the two different forms (1)

[3]These labels help understanding how the MAPE computations are distributed among the agents. For future scopes, these annotations may be extracted from the comments and used by the simulation environment at runtime.

and (2) depending on the *decentralized* or *centralized* control of the loop's computations, respectively.

$$\textbf{if } Cond \textbf{ then } Updates_K \qquad //@\text{M\_c[s]} \tag{1}$$

$$\textbf{if } Cond \textbf{ then } Analyze \qquad //@\text{M\_c[s]} \tag{2}$$

In both schemes, $Cond$, the condition under which the rule is applied, is an arbitrary first-order formula over monitored (in case of context-awareness) and/or controlled (in case of self-awareness) locations of the managed ASM. In the decentralized control scheme (1), $Updates_K$ is a finite set of transition rules simultaneously executed. They may consist of function updates $f(t_1, \ldots, t_n) := t$ changing (or defining, if there was none) the value of the knowledge location represented by the function f at the given parameters, and/or of call rules for more complex computations. Such knowledge updates may trigger an analyze activity executed by other managing agents. In case of centralized control (2), $Analyze$ is an ASM transition rule for an analyze computation (see schemes (3) and (4)) triggered by the monitoring computation and executed by the same agent in a waterfall style.

An analyze computation is specified by an ASM conditional rule annotated with @A:

$$\textbf{if } Cond_K \textbf{ then } Updates_K \qquad //@\text{A} \tag{3}$$

It involves the evaluation of a first order formula $Cond_K$, to determine if a violation of the system's goals occurs and an adaptation plan has to be triggered. This formula can be arbitrary complex and expresses the logic relationship of certain knowledge location values that must be true in order that violating situation holds. In a decentralized control, $Updates_K$ are updates of knowledge functions that may trigger planning activity executed by other agents. Alternatively, in centralized mode (schema 4), a planning computation can be directly executed by the same agent in a waterfall style.

$$\textbf{if } Cond_K \textbf{ then } Plan \qquad //@\text{A} \tag{4}$$

where $Plan$ is a transition rule for planning (see next section).

Note that complex planning computations may be missing in a MAPE-K loop and therefore the transition from an analyze computation to an execute computation may be direct. In this case, the ASM rules schemes 3 and 4 will trigger an execute activity indirectly (by knowledge updates) or directly (by executing an execute computation).

B. Adaptation Operators

A *plan computation* creates or selects a procedure to enact a necessary adaptation in the managed system. It can be a single action or a complex workflow. Then, as decided by the planning, an *execute computation* carries out the adaptation actions on the managed system using effectors.

In ASM, plan computations are ASM transition rules annotated with @P. They are conditional rules or may adopt a more complex ASM rule scheme. Such rules predispose the desired adaptation actions, and trigger (setting values of the shared knowledge) the managing agent(s) responsible to execute such adaptations or directly invoke execute computations.

Finally, an execute computation is an ASM rule annotated with @E and made of atomic adaptation actions. The following adaptation actions are supported:

- change locations value of the knowledge to (indirectly) trigger the execution of managed ASM agents playing the role of effectors;
- change locations value of the managed ASM (directly) by executing update rules or rule invocations (the effectors) of the managed ASM;
- stop/start managed ASM agents by setting the $program(a)$ of an agent a to, respectively, the *skip*-rule and a rule r;
- dynamically instantiate a new agent to introduce a new concurrent behavior by an *extend*-rule over the predefined domain *Agent*;
- dynamically change a component's agent behavior by updating the function $program(a)$ of an agent a to a new rule r.

C. Coordination

MAPE computations may be enhanced with support for distribution through coordination [36]. Cooperation and competition are forms of interactions among concurrent MAPE computations. So, interactive MAPE-K loops may require developing coordination models explicitly. To this purpose, ASM agents may adopt recurrent coordination patterns for distributed control (e.g., master-slaves pattern, hierarchical control pattern, alternate pattern, etc.) as formalized in [11].

VII. ASM Model of the Traffic Monitoring System

The ASM specification of a local camera system consists of four agents: *Camera* and *TrafficMonitor* representing the managed camera subsystem, and *OrganizationController* (*oc*) and *SelfHealingController* (*shc*) representing, respectively, the managing components *Organization Middleware* and *Self Healing Subsystem*.

We use the MAPE-K loops identified before to specify the self-adaptive behavior of the overall distributed system made of n cameras. The first MAPE-K loop deals with the flexibility concern to restructure organizations in case of congestion and it is handled by the organization controllers. The loop is defined as follows:

$$\textit{MAPE}(\textit{flexibility}) = \bigcup_{\mathtt{i}=1}^{n} \{\mathtt{orgContrFlexBehavior}[\mathtt{oc_i}]\}$$

where $\mathtt{orgContrFlexBehavior}[\mathtt{oc_i}]$ is an instance of the rule `orgContrFlexBehavior` for the organization controller $\mathtt{oc_i}$ of camera $\mathtt{i}$, for $\mathtt{i} = 1 \ldots n$.

The second MAPE-K loop deals with the adaptation to failures of other cameras (silent cameras). It is handled by both the organization controllers and the self-healing controllers. It is defined as:

$$\textit{MAPE}(\textit{extFailure}) = \bigcup_{\mathtt{i}=1}^{n} \left\{ \begin{array}{l} \mathtt{r\_failureAdapt}[\mathtt{oc_i}], \\ \mathtt{r\_failureDetect}[\mathtt{shc_i}] \end{array} \right\}$$

```
macro rule r_organizationController =
    par
        orgContrFlexBehavior(self) //Adaptation due to congestion
        r_failureAdapt[] //Adaptation due to external failure
        r_selfFailureAdapt[] //Adaptation due to internal failure
    endpar

agent OrganizationController : r_organizationController[]
```

Code 1: Program of each organization controller

```
macro rule r_selfHeal =
    par
        r_failureDetect[] //Adaptation due to external failure
        r_selfFailureDetect[] //Adaptation due to internal failure
    endpar

agent SelfHealingController: r_selfHeal[]
```

Code 2: Program of each self-healing controller

where `r_failureAdapt` is the rule of the organization controllers and `r_failureDetect` the rule of the self-healing controllers.

Finally, the third MAPE-K loop deals with internal failures of the camera. Both managing agents of each camera implement such a loop. It is defined as:

$$MAPE(intFailure) = \bigcup_{i=1}^{n} \left\{ \begin{array}{l} \texttt{r\_selfFailureAdapt}[\texttt{oc}_\texttt{i}], \\ \texttt{r\_selfFailureDetect}[\texttt{shc}_\texttt{i}] \end{array} \right\}$$

where `r_selfFailureAdapt` is the rule of the organization controllers and `r_selfFailureDetect` the rule of the self-healing controllers.

In the following, we describe the behavior of the two types of managing agents (i.e., the organization controller and the self-healing controller) that implement the three MAPE-K loops. We make use of the textual syntax `AsmetaL` of the ASMETA framework [7], [8]. The complete ASM specification is available online at [9].

An organization controller runs on each camera and is responsible for managing organization adaptations. Code 1 shows the organization controller's program[4] that executes in parallel three rules as contributions of the agent to the three MAPE-K control loops.

A self-healing controller runs on each camera. The corresponding ASM agent's program is the rule `r_selfHeal` reported in Code 2. It executes in parallel two rules for dealing with self-adaptation due to external and internal failures (the two MAPE-K loops for robustness) by alerting (by knowledge update) the organization controller that, in turn, will adapt the organization accordingly. This is the situation when two MAPE-K loops interact.

Details on the rule definitions can be found in the specification available on line. As exemplification, we here explain the last rule of the organization controller for adaptation due to internal failure. It is defined as instance of the (centralized) pattern (2) described in Sect. VI and it is partially shown in Code 3. This rule describes the contribution of the organization controller in the $intFailure$ MAPE-K loop. It consists of a *self-aware monitoring* computation (i.e., the concurrent reading of the probes `stopCam` and `startCam`) followed by an *analyze computation* (i.e., the reading of the knowledge's function `state` representing the camera's state). Finally, if required by the analyze computation, an *execute computation* is directly performed (i.e., the updating of the `state` of the camera) without a *plan* computation.

[4]Note that the concrete syntax **agent** `agent_type : rule[]` denotes in AsmetaL the initialization of the agent's function *program*.

```
macro rule r_selfFailureAdapt =
    par
        if stopCam(camera(self)) then //@M_s
            if state(camera(self)) != FAILED then //@A
                state(camera(self)) := FAILED //@E
            endif
        endif
        if startCam(camera(self)) then //@M_s
            if state(camera(self)) = FAILED then //@A
                par //@E
                    state(camera(self)) := MASTER
                    ...
                endpar
            endif
        endif
    endpar
```

Code 3: Excerpt of rule `r_selfFailureAdapt`

VIII. Validation and Verification

We here describe the model analysis activities we can perform on self-adaptive systems.

A. Validation

Model validation is a first model analysis activity, less demanding than property verification, that has to be used to validate the specification itself, but that can be also used to provide guarantees about qualities of the self-adaptive system. For model validation, we exploited the simulator `AsmetaS` [20] and the validator `AsmetaV` [16] of the toolset ASMETA [8].

1) Simulation: We performed either *interactive simulation*, where monitored inputs were provided interactively during simulation, and *random simulation*, where inputs values were chosen randomly by the simulator itself. During simulation, we also checked for *consistent updates*: in an ASM, two updates are inconsistent if they update the same location to two different values at the same time [12]. We discovered, by simulating (see the trace in Fig.5) a preliminary version of our specification (shown in Code 4), that the organization controller could, at the same time, turn the status of its camera both to `MASTER` and to `FAILED`. That particular situation could occur when a camera `c` was already `FAILED` and the system received the signals to both turn on and turn off the camera (i.e., both monitored locations `startCam(c)` and `stopCam(c)` were true). This was due to a wrong scheduling of the operations of the organization controller (the correct version is shown in Code 3).

By means of the `AsmetaS` simulator, we also checked for model *invariants*, namely properties that must hold in any

```
Insert a boolean constant for stopCam(c2):
true
Insert a boolean constant for startCam(c2):
false
...
<State 1 (controlled)>
state(c1)=MASTER
state(c2)=FAILED
state(c3)=MASTER
state(c4)=MASTER
</State 1 (controlled)>
Insert a boolean constant for stopCam(c2):
true
Insert a boolean constant for startCam(c2):
true
INCONSISTENT UPDATE FOUND:
location state(c2) updated to FAILED != MASTER
```

Fig. 5: Example of simulation – Detection of an inconsistent update

```
macro rule r_selfFailureAdapt =
    par
        if stopCam(camera(self)) then //@M_s
            state(camera(self)) := FAILED //@E
        endif
        if startCam(camera(self)) then //@M_s
            if state(camera(self)) = FAILED then //@A
                par //@E
                    state(camera(self)) := MASTER
                    ...
                endpar
            endif
        endif
    endpar
```

Code 4: Wrong version of rule `r_selfFailureAdapt`

```
scenario Flexibility_T0_T1
load main.asm

set stopCam(c1) := false; set stopCam(c2) := false; set stopCam(c3) := false;
set stopCam(c4) := false; set startCam(c1) := false; set startCam(c2) := false;
set startCam(c3) := false; set startCam(c4) := false; set congestion(c1) := false;
set congestion(c2) := false; set congestion(c3) := true; set congestion(c4) := true;
set elapsedWaitTime(shc3) := false; set elapsedWaitTimePlusDelta(shc4) := false;
exec par
        state(c3) := MASTERWITHSLAVES
        state(c4) := SLAVE
        slaves(c3, c4) := true
        getMaster(c4) := c3
        congested(oc3) := true
        congested(oc4) := true
    endpar;
step

set congestion(c2) := true;
step
check getMaster(c4)=c3 and s_offer(c3)=true and s_offer(c4)=false and
      slaves(c3,c4)=true and state(c1)=MASTER and state(c2) = MASTER and
      state(c3) = MASTERWITHSLAVES and state(c4)=SLAVE;

step
check isAlive(c4)=false and newSlave(c2,c3)=true and getMaster(c4)=c3 and
       s_offer(c3)=true and s_offer(c4)=false and slaves(c3,c4)=false and
       state(c1)=MASTER and state(c2) = MASTER and state(c3) = SLAVE and
       state(c4)=SLAVE;

step
check isAlive(c4) = false and newSlave(c2,c3) = false and getMaster(c4) = c3 and
      s_offer(c3) = true and s_offer(c4) = false and slaves(c2,c3) = true and
      slaves(c2,c4) = true and state(c1) = MASTER and
      state(c2) = MASTERWITHSLAVES and state(c3) = SLAVE and
      state(c4) = SLAVE;
```

Code 5: Flexibility validation scenario from T0 to T1 in Avalla

state of the machine execution. For example, we added to the specification the following invariants

(state(c_i) = FAILED and state(c_{i-1}) != FAILED) implies next(c_{i-1}) = c_{i+1}
(state(c_i) = FAILED and state(c_{i+1}) != FAILED) implies prev(c_{i+1}) = c_{i-1}

checking that the neighboring camera relations are correctly arranged after a failure: whenever a camera c_i fails (with $i = 2, \ldots, n-1$), camera c_{i-1} updates its next camera to c_{i+1}, and camera c_{i+1} updates its previous camera to c_{i-1}.

2) Scenario-based Validation: This technique consists in designing a set of scenarios specifying the expected behavior of the models. We simulated different scenarios with increasing number of cameras. In particular, we created and validated the adaptations scenarios shown in Fig. 2 from T0 to T1 for flexibility, and from T2 to T3 for adaptation to external failure.

Scenarios are expressed, by using constructs of the language `Avalla`, as interaction sequences of actor actions to *set* the environment (i.e., the values of monitored/shared functions), to *check* the machine state, to *ask* for the execution of given transition rules, and reactions of the machine which can perform a *step* or a sequence of *steps* until a condition holds. Scenarios are executed by the simulator `AsmetaS`, instrumented properly. During simulation, the validator `AsmetaV` captures any check violation and, if none occurs, it finishes with a *PASS* verdict.

The scenario reported in Code 5 describes the adaptation from T0 to T1. At the beginning, cameras `c3` and `c4` form an organization. When `c2` detects congestion, it joins the organization as MASTER. Appropriate assertions control that the right messages are sent and that the correct slavery relations are established.

B. Verification

We have verified the specifications through model checking. `AsmetaSMV` [4] is a tool of the ASMETA framework that translates ASM specifications into models of the NuSMV model checker. It allows the verification of *Computation Tree Logic* (CTL) and *Linear Temporal Logic* (LTL) formulae.

We have verified system-independent properties (or meta-properties), i.e., properties that any self-adaptive model should guarantee, and properties representing adaptation goals related to the requirements of the specific system. Sect. VIII-B1 presents the former category, while Sect. VIII-B2 describes the latter category.

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 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 [5] (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 *meta-properties* (*MP*, defined in [5] as CTL formulae). The violation of a

meta-property 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).

For this work, we have identified some meta-properties tailored for self-adaptive systems:

- MP_{nc}: MAPE loops are *not in conflict*. Two MAPE loops (or the same MAPE loop) are in conflict if they simultaneously update the same knowledge (i.e., the same location) at the same time to two different values. This definition corresponds to the definition of *inconsistent update* [12]. We have an *intra-loop* inconsistency if the inconsistent updates belong to the same MAPE loop, and an *inter-loop* inconsistency if the inconsistent updates belong to two different MAPE loops.
- MP_e: all rules involved in MAPE loops are *executed*. This meta-property only guarantees that there is no over specification inside a MAPE loop formalization. However, it does not guarantee functional correctness of a MAPE loop. This can only be controlled with application-dependent properties specified by the user, as described in Sect. VIII-B2.
- MP_m: the knowledge is *minimal*, i.e., it does not contain locations that are *unnecessary* (they are never read nor updated) or that do no assume all the values of their codomains. Note that a violation of this meta-property may also indicate that the specification is not complete, i.e., that the designer forgot to read/update a location.

In our first developed models we discovered an intra-loop inconsistency (meta-property MP_{nc}) caused by the inconsistent updates also found by simulation (see Sect. VIII-A1). Although a normal simulation or the scenario-based validation can sometimes unveil the presence of inconsistent updates, when the model becomes particularly complex, inconsistencies may be more difficult to find, and an automatic approach as that provided by the model reviewer is helpful. Moreover, simulation can show only *some* inconsistencies (i.e., those detected in the executed runs), whereas model review detects *all* the inconsistencies.

We have also found several minimality violations (i.e., meta-property MP_m), since some locations could not assume all the values of their codomains. For example, the boolean binary function `slave(Camera, Camera)` represents the slavery relations exiting between the cameras: location `slave(`c_i`,` c_j`)` is true iff c_j is slave of c_i. The model reviewer advised us that all the locations `slave(`c_i`,` c_j`)` with $j \leq i$ cannot be true: indeed, a camera cannot be slave of a subsequent camera or of itself. Obviously, in this case the meta-property violation does not indicate a real fault, but only that the model is not minimal. In order to address the minimality violation, we could have modeled the slavery relations using a different data structure (e.g., associating with each camera the set of its slaves). Note that, although in the formal specification the chosen data structure is not a real issue, in the final implementation the choice of an optimized data structure could be important, particularly if there are some memory limitations on the hardware.

2) Verification of Case Study Requirements: Model review permits to verify general properties (automatically built from the model) that any model should guarantee. More complicated properties related to the requirements of the application must be specified by the modeler.

We have verified classical temporal properties to guarantee correctness and reliability of our running case study. Moreover, we also considered properties originally proposed in [24], dividing them in three categories: invariants, flexibility, and robustness to silent node failures. Below we report some of the specified properties.

Invariants: These properties must hold in all the states. For example, we have verified that all cameras cannot be slaves (or master with slaves) at the same time.

I1: ag(not(**forall** $c **in** Camera **with** state($c) = SLAVE))
I2: ag(not(**forall** $c **in** Camera **with** state($c) = MASTERWITHSLAVES))

Note that the `AsmetaS` simulator supports invariant checking. Each invariant φ, here verified through model checking with the temporal property $\mathbf{ag}(\varphi)$, has also been checked by the simulator and the scenario-based validator. Obviously, by simulation we have been able to verify only the states covered by the executed runs, whereas model checking gave us the assurance that the invariants hold in each model state.

Flexibility: The following properties check that the system correctly adapts itself to the different traffic conditions (i.e., presence or absence of traffic congestion). We have proved different properties, considering the possible different roles assumed by the cameras when they observe congestion.

For example, we have checked the most basic organization: when a master camera c_i detects a congestion and the next camera c_{i+1} is master and congested as well, then the two cameras form an organization where c_i is master with slaves and c_{i+1} its slave (for each $i = 1, \ldots, n-1$).

F1: ag((state(c_i) = MASTER and congested(oc_i) and
state(c_{i+1}) = MASTER and congested(oc_{i+1})) implies
af(state(c_i) = MASTERWITHSLAVES and slaves(c_i, c_{i+1})))

Based on our master election policy, we have checked more complex properties, as, for example, the following. If three consecutive master cameras are congested, they form an organization, where the leftmost camera c_i is the master and the other two cameras are its slaves (with $i = 2, \ldots, n-2$).

F4: ag((state(c_i) = MASTER and congested(oc_i) and
state(c_{i+1}) = MASTER and congested(oc_{i+1}) and
state(c_{i+2}) = MASTER and congested(oc_{i+2}) and
stateC(c_{i-1}) = MASTER and not(congested(oc_{i-1}))) implies
ef(stateC(c_i) = MASTERWITHSLAVES and slaves(c_i, c_{i+1}) and
slaves(c_i, c_{i+2})))

Robustness: We have verified that the system is able to correctly recover from a silent node failure, i.e., that the non-failing cameras reorganize themselves correctly.

First, we have checked that, if a camera c_i fails (being c_{i+1} its slave), then c_{i+1} leaves its master (with $i = 1, \ldots, n-1$).

R1: ag((stateC(c_i) = FAILED and slaves(c_i, c_{i+1})) implies
ef(not(slaves(c_i, c_{i+1}))))

As additional property we have checked that, whenever the slaves of an organization detect a failure of their master, they eventually form a new organization, as long as the traffic remains congested. We only show one configuration, but all the possible configurations have been checked.

R2: ag((stateC(c1) = FAILED and slaves(c1,c2) and slaves(c1,c3))
implies e[(congested(orgCont2) and congested(oc3)) U
(stateC(c2) = MASTERWITHSLAVES and slaves(c2,c3))])

IX. Discussion & Faced Challenges

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.

Modeling self-adaptation features was possible thanks to the multi-agent computational model available in ASMs to specify distributed computation and coordination among agents. Furthermore, in the way we model MAPE-K control loops in ASMs, we achieve a clear separation between adaptation logic and functional logic. This is possible since the formal approach allows:

(i) To separate, by modeling them as separated agents, managing components from managed ones. E.g., the agent *OrganizationController* manages the local camera.

(ii) To distribute a MAPE loop among those agents that are involved in the loop's computations. E.g., both agents *SelfHealingController* and *OrganizationController* are involved in the robustness loops for external and internal failure.

(iii) To separate, inside the behavior of a managing agent, different adaptation concerns (modeled by separated transition rules). E.g., in the program of the *OrganizationController*, the rule `orgControlFlexBehaviour` models the agent's contribution in the flexibility concern, while the rule `r_failureAdapt` is related to the robustness concern for external failures.

(iv) To distinguish between decentralized and centralized loop's control, by defining specific rule schemes.

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 also facilities reasoning about components behavior and avoid over-specification, keeping models concise.

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

(v) *Validate adaptation requirements by executing specifications*. For the Traffic Monitoring case study, we simulated different scenarios for flexibility and robustness.

(vi) *Determine conflicting MAPE loops*. Simultaneous execution of different adaptive behaviors might cause consistency violations, when different MAPE loops update locations inconsistently. Model simulation, by checking for inconsistent updates, can help to discover these situations. However, a deeper model analysis, by means of the proof of suitable meta-properties, can reveal intra/inter-loop inconsistencies inside the agents' programs. Such conflicting situations often requires to reason about "priorities" of adaptation concerns, which can be established by appropriate scheduling of the agents' operations. In our case study, by this technique, we discovered an intra-loop inconsistency inside the program of the *OrganizationController*.

(vii) *Assert the system correctness*. Once one gains enough confidence that the self-adaptive system works according to the expected adaptation logic, verification is necessary to assure correctness properties. By model checking, we have verified a set of properties expressing, as temporal logic formulas, adaptation goals (mainly flexibility and robustness) of our case study.

(viii) *Check for model completeness without over-specification*. Besides correct, a model must be complete, i.e., all the computational and adaptive aspects must be specified, but also minimal, i.e., no over-specification must be, due to unnecessary signature and/or transition rules never executed. We checked suitable meta-properties to guarantee knowledge minimality and the absence of "dead" rules. In our case study we found some minimality violations.

X. Conclusion and Future Directions

In this paper, we showed how to model the self-adaptation layer of modern distributed systems in terms of multi-agent ASMs. By exploiting the abstraction and flexibility of this formal method, we introduced the concept of self-adaptive ASMs as formal framework to specify self-adaptive behavior. We applied our formal modeling approach to the Traffic Monitoring case study. We were able to avoid over-specification while achieving clear separation of concerns that helped us to focus on one adaptation concern at a time, and, for each concern, to separate the managing behavior from the managed one. We were able to model the interaction between managed and managing agents as ASM agents interaction, without the necessity to extend the ASM formalism. Validation and verification activities helped us to reason about *interfering* adaptation concerns and adaptation goals.

In the future, we want to exploit runtime monitoring techniques, that the ASM formalism already supports [6] for Java-like programs, to connect our formal model to a runtime adaptation middleware. We also plan to exploit appropriate extensions of ASMs with time models [22] for specifying time-triggered adaptation.

Acknowledgment

This work was supported in part by the Italian Ministry of Research within the PRIN project "GenData 2020".

References

[1] D. B. Abeywickrama, N. Hoch, and F. Zambonelli. SimSOTA: engineering and simulating feedback loops for self-adaptive systems. In B. C. Desai, A. M. de Almeida, J. Bernardino, and S. P. Mudur, editors, *International C* Conference on Computer Science & Software*

Engineering, C3S2E13, Porto, Portugal - July 10 - 12, 2013, pages 67–76. ACM, 2013.

[2] D. B. Abeywickrama and F. Zambonelli. Model checking goal-oriented requirements for self-adaptive systems. In M. Popovic, B. Schätz, and S. Voss, editors, *IEEE 19th International Conference and Workshops on Engineering of Computer-Based Systems, ECBS 2012, Novi Sad, Serbia, April 11-13, 2012*, pages 33–42. IEEE, 2012.

[3] R. Adler, I. Schaefer, T. Schüle, and E. Vecchié. From model-based design to formal verification of adaptive embedded systems. In M. Butler, M. G. Hinchey, and M. M. Larrondo-Petrie, editors, *Formal Methods and Software Engineering, 9th International Conference on Formal Engineering Methods, ICFEM 2007, Boca Raton, FL, USA, November 14-15, 2007, Proceedings*, volume 4789 of *Lecture Notes in Computer Science*, pages 76–95. Springer, 2007.

[4] P. Arcaini, A. Gargantini, and E. Riccobene. AsmetaSMV: A Way to Link High-Level ASM Models to Low-Level NuSMV Specifications. In M. Frappier, U. Glässer, S. Khurshid, R. Laleau, and S. Reeves, editors, *Abstract State Machines, Alloy, B and Z*, volume 5977 of *Lecture Notes in Computer Science*, pages 61–74. Springer Berlin Heidelberg, 2010.

[5] P. Arcaini, A. Gargantini, and E. Riccobene. Automatic review of abstract state machines by meta property verification. In C. Muñoz, editor, *Proceedings of the Second NASA Formal Methods Symposium (NFM 2010), NASA/CP-2010-216215*, pages 4–13, Langley Research Center, Hampton VA 23681-2199, USA, April 2010. NASA.

[6] P. Arcaini, A. Gargantini, and E. Riccobene. CoMA: Conformance Monitoring of Java Programs by Abstract State Machines. In S. Khurshid and K. Sen, editors, *Runtime Verification*, volume 7186 of *Lecture Notes in Computer Science*, pages 223–238. Springer Berlin Heidelberg, 2012.

[7] P. Arcaini, A. Gargantini, E. Riccobene, and P. Scandurra. A model-driven process for engineering a toolset for a formal method. *Softw., Pract. Exper.*, 41(2):155–166, 2011.

[8] The ASMETA toolset website. http://asmeta.sourceforge.net/, 2011.

[9] http://svn.code.sf.net/p/asmeta/code/asm_examples/DAS/TrafficMonitoring-System/seams2015/, 2015.

[10] B. Bartels and M. Kleine. 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, pages 158–167, New York, NY, USA, 2011. ACM.

[11] E. Börger. Modeling workflow patterns from first principles. In C. Parent, K.-D. Schewe, V. C. Storey, and B. Thalheim, editors, *ER*, volume 4801 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2007.

[12] E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, 2003.

[13] Y. Brun, G. D. M. Serugendo, C. Gacek, H. Giese, H. M. Kienle, M. Litoiu, H. A. Müller, M. Pezzè, and M. Shaw. Engineering self-adaptive systems through feedback loops. In B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, editors, *Software Engineering for Self-Adaptive Systems [outcome of a Dagstuhl Seminar]*, volume 5525 of *Lecture Notes in Computer Science*, pages 48–70. Springer, 2009.

[14] R. Bruni, A. Corradini, F. Gadducci, A. Lluch-Lafuente, and A. Vandin. Adaptable transition systems. In N. Martí-Oliet and M. Palomino, editors, *Recent Trends in Algebraic Development Techniques, 21st International Workshop, WADT 2012, Salamanca, Spain, June 7-10, 2012, Revised Selected Papers*, volume 7841 of *Lecture Notes in Computer Science*, pages 95–110. Springer, 2012.

[15] R. Bruni, A. Corradini, F. Gadducci, A. Lluch-Lafuente, and A. Vandin. Modelling and analyzing adaptive self-assembly strategies with maude. In F. Durán, editor, *Rewriting Logic and Its Applications - 9th International Workshop, WRLA 2012, Held as a Satellite Event of ETAPS, Tallinn, Estonia, March 24-25, 2012, Revised Selected Papers*, volume 7571 of *Lecture Notes in Computer Science*, pages 118–138. Springer, 2012.

[16] A. Carioni, A. Gargantini, E. Riccobene, and P. Scandurra. A Scenario-Based Validation Language for ASMs. In E. Börger, M. Butler, J. P. Bowen, and P. Boca, editors, *Abstract State Machines, B and Z*, volume 5238 of *Lecture Notes in Computer Science*, pages 71–84. Springer Berlin Heidelberg, 2008.

[17] B. H. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, et al. *Software engineering for self-adaptive systems: A research roadmap*. Springer, 2009.

[18] R. De Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. M. Villegas, T. Vogel, et al. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*, pages 1–32. Springer, 2013.

[19] B. Djoudi, C. Bouanaka, and N. Zeghib. Model checking pervasive context-aware systems. In S. Reddy, editor, *2014 IEEE 23rd International WETICE Conference, WETICE 2014, Parma, Italy, 23-25 June, 2014*, pages 92–97. IEEE, 2014.

[20] A. Gargantini, E. Riccobene, and P. Scandurra. A Metamodel-based Language and a Simulation Engine for Abstract State Machines. *J. Universal Computer Science*, 14(12):1949–1983, 2008.

[21] I. Georgiadis, J. Magee, and J. Kramer. Self-organising software architectures for distributed systems. In D. Garlan, J. Kramer, and A. L. Wolf, editors, *WOSS*, pages 33–38. ACM, 2002.

[22] S. Graf and A. Prinz. Time in state machines. *Fundam. Inform.*, 77(1-2):143–174, 2007.

[23] M. Güdemann, A. Angerer, F. Ortmeier, and W. Reif. Modeling of self-adaptive systems with SCADE. In *International Symposium on Circuits and Systems (ISCAS 2007), 27-20 May 2007, New Orleans, Louisiana, USA*, pages 2922–2925. IEEE, 2007.

[24] M. U. Iftikhar and D. Weyns. A case study on formal verification of self-adaptive behaviors in a decentralized system. In Kokash and Ravara [27], pages 45–62.

[25] M. U. Iftikhar and D. Weyns. Activforms: active formal models for self-adaptation. In G. Engels and N. Bencomo, editors, *SEAMS*, pages 125–134. ACM, 2014.

[26] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

[27] N. Kokash and A. Ravara, editors. *Proceedings 11th International Workshop on Foundations of Coordination Languages and Self Adaptation, FOCLASA 2012, Newcastle, U.K., September 8, 2012*, volume 91 of *EPTCS*, 2012.

[28] A. Lanoix, J. Dormoy, and O. Kouchnarenko. Combining proof and model-checking to validate reconfigurable architectures. *Electronic Notes in Theoretical Computer Science*, 279(2):43 – 57, 2011.

[29] J. Magee and T. Maibaum. 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, pages 30–36, New York, NY, USA, 2006. ACM.

[30] E. Merelli, N. Paoletti, and L. Tesei. A multi-level model for self-adaptive systems. In Kokash and Ravara [27], pages 112–126.

[31] E. Riccobene and P. Scandurra. Towards ASM-based formal specification of self-adaptive systems. In Y. A. Ameur and K. Schewe, editors, *ABZ*, volume 8477 of *Lecture Notes in Computer Science*, pages 204–209. Springer, 2014.

[32] E. Riccobene and P. Scandurra. Formal modeling self-adaptive service-oriented applications. In *Proc. ACM SAC 2015, 30th ACM Symposium on Applied Computing, Service-Oriented Architecture and Programming (SOAP) Track*, 2015.

[33] P. Vromant, D. Weyns, S. Malek, and J. Andersson. On interacting control loops in self-adaptive systems. In *Software Engineering for Self-Adaptive Systems, SEAMS 2011*, 2011.

[34] D. Weyns, M. U. Iftikhar, D. G. de la Iglesia, and T. Ahmad. A survey of formal methods in self-adaptive systems. In B. C. Desai, E. Vassev, S. P. Mudur, and B. C. Desai, editors, *C3S2E*, pages 67–79. ACM, 2012.

[35] D. Weyns, S. Malek, and J. Andersson. FORMS: a formal reference model for self-adaptation. In M. Parashar, R. J. O. Figueiredo, and E. Kiciman, editors, *ICAC*, pages 205–214. ACM, 2010.

[36] D. Weyns, B. R. Schmerl, V. Grassi, S. Malek, R. Mirandola, C. Prehofer, J. Wuttke, J. Andersson, H. Giese, and K. M. Göschka. On patterns for decentralized control in self-adaptive systems. In R. de Lemos, H. Giese, H. A. Müller, and M. Shaw, editors, *Software Engineering for Self-Adaptive Systems*, volume 7475 of *Lecture Notes in Computer Science*, pages 76–107. Springer, 2010.

[37] J. Zhang and B. H. C. Cheng. Model-based development of dynamically adaptive software. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 371–380, New York, NY, USA, 2006. ACM.