Modeling and Validating
Self-adaptive Service-oriented Applications

Paolo Arcaini
Charles University in Prague
Faculty of Mathematics and Physics, Czech Republic
arcaini@d3s.mff.cuni.cz

Elvinia Riccobene
Dept. of Computer Science
Università degli Studi di Milano, Italy
elvinia.riccobene@unimi.it

Patrizia Scandurra
Dept. of Management, Information and Production Engineering
Università degli Studi di Bergamo, Italy
patrizia.scandurra@unibg.it

ABSTRACT
Self-adaptive and autonomous behaviors are becoming more and more important in the context of service-oriented applications, and formal modeling self-adaptive service-oriented components is highly required to assure quality properties.

This paper enhances the formal framework SCA-ASM for modeling and validating distributed self-adaptive service-oriented applications. We explain how modeling an SCA-ASM component able to monitor and react to environmental changes (context-awareness) and to internal changes (self-awareness), and present the operators for expressing and coordinating self-adaptive behaviors in a distributed setting. We also support techniques for validating adaptation scenarios, and getting feedback of the correctness of the adaptation logic as implemented by the managing SCA-ASM components over the managed ones. As a proof-of-concepts, we use self-adaptive SCA-ASMs for modeling and validating a decentralized traffic monitoring system.

CCS Concepts
• General and reference → Validation; • Software and its engineering → Formal methods; Requirements analysis;

Keywords
Self-adaptation; service-oriented applications; formal modeling; validation; SCA-ASM

1. INTRODUCTION
Service-oriented applications are playing so far an important role in several application domains (e.g., information technology, health care, robotics, defense and aerospace, to name a few). Cloud service providers, in particular, are expanding their offerings to include the entire traditional IT stack, ranging from foundational hardware and platforms to application components, software services, and whole software applications.

Self-adaptation (SA) [12, 14, 22] is recognized as an effective approach to deal with the increasing complexity, uncertainty and dynamicity of modern service-oriented applications where service components can appear and disappear, may become temporarily or permanently unavailable, may continuously change their behavior. A self-adaptive component is able to autonomously adapt its behavior to achieve required goals and functionality, on the basis of its internal dynamics and changing conditions in the environment.

To assure the required quality properties (e.g., flexibility, robustness, etc.), techniques for modeling and reasoning about adaptation capabilities of service-oriented components and their assemblies (or compositions) are greatly in demand, both at design time and at runtime. However, the current answer to this request is very limited [34], and this work would be a step forward along this research line.

Formal frameworks for modeling service-oriented applications already exist. Among them, SCA-ASM [28, 26] is based on the combined use of the OASIS standard SCA (Service Component Architecture) [29] to model the architecture and the assembly of the application, and the ASM (Abstract State Machine) formal method [8] to specify services’ behavior. The SCA-ASM language provides modeling primitives to represent service component assemblies, to express internal service computation, interaction and orchestration, as well as fault and compensation handling.

By adopting the decentralized MAPE-K control loop model of SA [22], we here show how SCA-ASM can be used to formalize and execute components exposing self-adaptive behavior in a decentralized manner and with partial shared knowledge. In particular, we explain how modeling in an SCA-ASM component its capacity to cope with continuously changing conditions of the environment (context-awareness), and to react to internal changes (self-awareness); we introduce operators for expressing and coordinating self-adaptive behavior in a distributed setting.

SCA-ASM enhanced with modeling features for SA provides a formal framework for the rigorous development of self-adaptive service-oriented applications. The proposed framework offers several advantages with respect to the current state of art (see Sect. 8 for related works). SCA-ASM allows modeling both structure and behavior of service com-
ponents in a unique framework integrating architectural and behavioral views. Based on the practical and scientifically well-founded ASM formal method, SCA-ASM models are executable and without mathematical overkill. The framework supports the design principle of separation of concerns and different adaptive behaviors can be introduced separately and in an incremental way, discovering possible side effects. Finally, by exploiting the prototyping/simulation environment for SCA-ASM [26], components can be executed already at high level of formalization, without caring about implementation details. Early validation by model simulation is a great means for evaluating architectural choices and alternative designs with limited implementation effort. Moreover, the mathematical foundation of the method facilitates reasoning about component behavior in order to guarantee correctness of its adaptation logic on the basis of the adaptation concerns.

A first overview of the self-adaptive SCA-ASM framework was presented in [27]. This article extends the earlier work in [27] by improving and extending the description of the modeling framework (modeling constructs and patterns) for the specification of self-adaptive service components with decentralized adaptation control. Through a case study, we also describe here the formal techniques supported by our framework for validating adaptation scenarios and getting feedback (already at system design time) of the correctness of the adaptation logic. Formal validation of quality properties is possible in terms of model simulation [17] and construction of execution scenarios [11]. We here mainly focus on flexibility (i.e., the ability of the system to dynamically adapt to changing conditions in the environment) and robustness (i.e., the ability of the system to cope autonomously with errors during execution).

This paper is organized as follows. Sect. 2 briefly introduces a reference model for SA. Sect. 3 describes the running case study of a Traffic Monitoring Application (TMA). Sect. 4 provides a background on the SCA-ASM modeling language for service oriented applications. Sect. 5 shows how to model SA in SCA-ASM in a decentralized manner. Sect. 6 reports, as case study, the results of modeling the architecture and the behavior of the TMA example. Sect. 7 presents the results of the TMA SCA-ASM model validation by simulating adaptation scenarios. Sect. 8 presents contributions related to our work. Sect. 9 discusses some lessons learned, while Sect. 10 concludes the paper and outlines future research directions of our work.

2. REFERENCE MODEL FOR SA

According to the reference model FORMS [35] 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, 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) [22] 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 compiles 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.

3. RUNNING CASE STUDY

As case study to illustrate the proposed modeling and validation framework, we adopt the Traffic Monitoring Application (TMA) presented in [20, 33]. It is an example of decentralized adaptation control in the application domain of traffic monitoring.

A number of intelligent cameras are distributed along a road (see Fig. 2). A camera is able to measure the traffic conditions and decide whether there is a traffic jam within its viewing range. A camera is endowed with a data processing unit and a communication unit to interact with other cameras. Traffic jams can span the viewing range of multiple cameras and can dynamically grow and dissolve. The dynamic nature of the traffic requires dynamic cameras collaborations. Since there is no central control, cameras have to collaborate in organizations to observe larger phenom-
Figure 2: Adaptation scenarios (adapted from [20])

Figure 3: SCA-ASM component shape

4. BACKGROUND ON SCA-ASM

The SCA-ASM language has been defined [28, 26] for modeling service-oriented applications. It complements the SCA component model with the ASM (Abstract State Machine) model of computation to provide an ASM-based formal and executable description of services internal behavior, orchestration and interactions.

An SCA-ASM service-oriented component appears as depicted in Fig. 3. It consists of an SCA graphical front-end to model the structure of the component, and an ASM model to specify the component’s behavior.

The SCA model (Component A) graphically represents: the services (AService) provided by the component, the references (b) (functions required by the component) wired to services provided by other components, the properties (pA) allowing for the configuration of a component implementation and bindings that specify access mechanisms used by services and references according to some technology/protocol. Services and references are typed by interfaces.

The ASM model, given in terms of ASM modules – one for each service interface (module AService), and one to describe the component (module A) – is used to express how a component behaves to provide its own services and to interact with the other components.

An ASM is an extension of a FSM [8] 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. Formally speaking, an ASM state is a multi-sorted first-order structure, i.e., domains of objects with functions and predicates 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 SCA-ASM language exploits the distributed multi-agents ASM computational model where a number of running agents may interact in a synchronous or asynchronous way.

An SCA-ASM service-oriented component is endowed with (at least) one ASM agent computing according with the component’s business partner (or role). Each agent a executes its own, possibly the same but differently instantiated, program(a)3 that specifies the agent’s behavior and the services provided by the component in terms of ASM transition rules. Components’ agents are able to interact with other agents by providing/requiring services to/from other service-oriented components’ agents.

The interface module of an SCA-ASM component A (see the skeleton reported on the left of Fig. 3) declares a dynamic controlled function s that returns a service as a rule. Such a service function represents a provided service rule since, once initialized, it leads to binding each invocation of s (read: application of function s) to a service rule r_s of the component. It can be dynamically updated (at runtime) as a built-in service adaptation mechanism (see Sect. 5.2).

In the ASM module for components (see the skeleton reported on the right of Fig. 3), the annotations are used to denote SCA concepts (i.e., references, properties, services, etc.). Both ASM modules are expressed by using the textual notation ASMETA/AsmetaL2.

1 In ASM, a function program on Agent indicates the named transition rule associated with an agent as its behavior. It is used to dynamically associate and change behavior to agents.

2 Two grammatical conventions must be recalled: a variable identifier starts with $; a rule identifier begins with “r_”.

2
ASM rule constructors and predefined ASM rules (i.e., named ASM rules made available as model library) are used as basic SCA-ASM behavioral primitives. They are reported in Table 1 and include: rule constructors to specify basic computation actions (location update, service invocation, or do nothing) and coordination such as conditional actions (if-then-else), parallel actions (par), sequential actions (seq), iterations (iterate, while, recvwhile), actions to create child agents (spawn), non-determinism (existential quantification choose) and unrestricted synchronous parallelism (universal quantification forall), etc.; communication rules — send, receive, send-receive and reply — providing synchronous/asynchronous service invocation styles (corresponding, respectively, to the request-response and one-way interaction patterns of the SCA standard). Communication relies on a dynamic domain Message that represents message instances managed by an abstract message-passing mechanism: components communicate over wires and a message encapsulates information about the partner link and the referenced service name and data transferred.

SCA-ASM rule constructors can be combined to model specific interaction and orchestration patterns in well structured and modularized entities.

SCA-ASM modeling constructs for fault/compensation handling are also supported (see [28, 26]), but are not reported here since related to fault tolerance concepts that we do not take into account in the adaptation model presented here.

An SCA-ASM design environment [30, 9, 26] was developed by integrating the Eclipse-based SCA Composite Designer, the SCA runtime platform Tuscany [32], and the simulator ASMETA/AsmetaS [17, 4, 5]. This environment allows a designer to graphically model, compose, deploy, and execute heterogeneous service-oriented applications in a technology-agnostically way.

5. SELF-ADAPTIVE SCA-ASM

Here we introduce the concept of self-adaptive SCA-ASM assembly to model distributed self-adaptive service-oriented applications from the perspective of distributed and interacting MAPE-K feedback control loops.

According to the reference model for SA in Sect. 2, we distinguish managed SCA-ASM components encapsulating the functional logic of the application and managing SCA-ASM components encapsulating the logic of adaptation.

A managed SCA-ASM component is an SCA-ASM service component as presented in Sect. 4.

A managing SCA-ASM component is, instead, a self-adaptive SCA-ASM component endowed, besides the functionalities of an SCA-ASM service component, with mechanisms/operators to monitor the environment and itself, and to perform adaptation actions.

The assembly of the self-adaptive components should expose a certain number of MAPE-K loops

\[ \{MAPE(adj_1), \ldots, MAPE(adj_k)\} \]

one per each adaptation concern adj_i. The four computations of each MAPE(adj_i) are formalized in terms of ASM transition rules distributed among those managing SCA-ASM components involved in the execution of the loop.

Therefore, the program of an agent a associated to a managing SCA-ASM component has the form

\[ \text{program}(a) = \bigcirc \left( R_{MAPE(adj_j)}^a \right)_{j=1 \ldots k} \]

where the operator \( \bigcirc \) is an SCA-ASM coordination rule constructor (most of the times par or seq for parallel or sequential actions coordination), and rules \( R_{MAPE(adj_j)}^a \) specify the behavioral contributions of the agent a to the k loops the component is involved in.

These rules are annotated (as comments //) with appropriate labels @M (for context-aware monitoring), @L (for self-aware monitoring), @A (for analyzing), @P (for planning), and @E (for execution), depending on the role of the agent a in the loop. These labels (better explained in the following subsections) show how MAPE computations are distributed among the agents. For reasoning and simulation scopes, these annotations may be extracted from comments and used by a runtime platform.

An adaptation concern adj is also characterized by a knowledge K(adj). Therefore, further SCA-ASM components represent the knowledge of the MAPE loops. These components are used by the managing components as reflective model to save and share data (through suitable access services) of the
managed subsystem and the environment, and other information relevant for the MAPE computations. In principle we can have a Knowledge component for each MAPE(adj). However, it is possible to collapse more knowledge components into one (especially when more MAPE-K loops are performed by the same managing component).

The notion of environment is directly supported in SCA-ASM through monitored functions\(^8\) of the ASM theory \([8]\). Probes are represented by actions of managed components’ agents and consist into reading and reporting values of monitored functions. Similarly, actuators are actions of managed agents that update the value of controlled functions\(^8\) on the base of the plan decided by the managing agents.

In the rest of this section, we describe how the self-adaptive SCA-ASM component model realizes the following key requirements for SA (see the reference model FORMS \([35]\)):

- How a component system monitors the environment and itself;
- How a component system adapts itself;
- How to coordinate monitoring and adaptation in a distributed setting.

### 5.1 Monitor and analyze computations

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 of a MAPE-K loop assesses the collected data from the environment and/or the system itself to determine the system’s ability to satisfy its objectives. Update computations in combination with analyze computations provide 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 of self-adaptive systems.

In SCA-ASM, we prefer 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 (or portions of rules) annotated with \(@M_c\) and \(@M_s\), respectively. Context-aware monitoring consists of looking at values of ASM monitored locations (the environment), while self-aware monitoring consists of looking at values of ASM controlled locations (internal locations) of the managed system.

Context/self-aware monitoring computations may have two different rule schemes (1 and 2) depending on the decentralized or centralized control of the loop’s computations, respectively:

\[
\text{if } \text{Cond} \text{ then } \text{Updates}_K \quad //@M_c[s] \quad (1)
\]

\[
\text{if } \text{Cond}_K \text{ then } \text{Plan} \quad //@A \quad (2)
\]

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 SCA-ASM. In the decentralized control scheme (1), \(\text{Update}_K\) 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. Knowledge updates have to be implemented in concrete terms as invocations of appropriate access services of the knowledge component of the MAPE-K loop. Such knowledge updates (i.e., once the corresponding services executions complete) may then trigger an analyze activity executed by other managing components’ agents. In case of centralized control (2), Analyze is an ASM transition rule for an analyze computation (see schemes 3, 4) triggered by the monitoring computation and executed by the same component’s agent in a waterfall style.

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

\[
\text{if } \text{Cond}_K \text{ then } \text{Update}_K \quad //@A \quad (3)
\]

It involves the evaluation of a first order formula \(\text{Cond}_K\), 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, \(\text{Update}_K\) are updates of knowledge functions that may trigger planning activity executed by other components’ agents. Alternatively, in centralized mode (schema 4), a planning computation can be directly executed by the same component’s agent in a waterfall style:

\[
\text{if } \text{Cond}_K \text{ then } \text{Plan} \quad //@A \quad (4)
\]

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).

### 5.2 Plan and execute computations

In a MAPE-K loop, analyze computations may trigger plan computations when a particular analysis discovers a violation of the system’s objectives. A plan computation creates/selects a procedure to enact a necessary adaptation in the managed system. The plan computation can be a single action or a complex workflow. Then, as decided by the plan computation, an execute computation carries out the adaptation actions on the managed system using effectors.

In SCA-ASM, plan computations are executed by managing agents and consist of ASM rules annotated with \(@P\). These rules can be conditional rules or may adopt a more complex...
ASM rule scheme. These rules determine the desired adaptation actions and trigger (by sending information or setting values of the shared knowledge) the managing agent(s) responsible to execute such adaptations or directly invoke execute computations.

An execute computation is an ASM rule annotated with @E and usually made of atomic adaptation actions. In SCA-ASM, a set of atomic adaptation actions allow runtime adaptation to be expressed both at architectural and at behavioral level.

Structural adaptation actions.
- add/remove components to/from a composite;
- add/remove component services, references, properties to/from a component;
- add/remove wires to bind/unbind reference-service interfaces;
- add/remove wires to delegate/undelegate service-service and reference-reference interfaces.

These architectural re-configurations are effectively executed by an SCA runtime platform with dynamic introspection and adaptation capabilities (e.g., the FraSCAti platform [31]).

Behavioral adaptation actions.
- change values of managed component’s properties (by firing update rules);
- change the knowledge (by invoking appropriate services of the knowledge);
- stop/start components (by updating the function status of a component’s life cycle to the value INIT/EXITED [26]);
- instantiate a new agent within a component to introduce a new concurrent behavior (by a spawn rule);
- change a component’s agent behavior dynamically (by updating the function program(a) of agent a to a new rule r);
- change a component’s service behavior dynamically (by updating the service function s(a) of an agent a to a new service rule r’).

@E can also label non-atomic SCA-ASM rules modeling exception or compensation handlers [26] to be executed, respectively, in case of fault or to rollback partially executed actions.

5.3 Distributed coordination
Distributed adaptive service-oriented applications are specified as self-adaptive SCA-ASM assembly that is the composition of managed SCA-ASM components, managing self-adaptive SCA-ASM components, plus SCA-ASM components for the shared MAPE-K loops’ knowledge.

Figure 4: SCA model: Camera Managing Subsystem
The computational model is that of a multi-agent ASM where multiple components’ agents interact in parallel in a synchronous/asyncronous way. Moreover, 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, SCA-ASM components’ agents may adopt recurrent coordination patterns for distributed control (e.g., master-slaves pattern, hierarchical control pattern, alternate pattern, etc.) as formalized in [7].

6. SCA-ASM MODEL OF THE TMA
In this section we exemplify the self-adaptive SCA-ASM assembly by our running case study.

6.1 System architecture
Fig. 4 shows the SCA architecture5 of the managing subsystem of each camera system as adapted from [20]. According to the vision provided in Sect. 2, the managed subsystem is the local camera, while the managing subsystem corresponds essentially to the components Self-healing Controller and Organization Controller. The local camera (represented as an external reference cam) provides the functionality to detects traffic jams and inform clients. Traffic detection is carried out by the camera through a sensor that detects traffic jam and in case of “congestion” or “no longer congestion” notifies the organization controller (see the service interface TrafficNotifyService in Fig. 4).

Cameras collaboration is managed by organization controllers using a master/slave control model. For each organization

5The case study presents Event-Driven interactions not yet supported in the SCA runtime Tuscany. We model events in SCA-ASM as asynchronous service requests. An event type is defined in the component implementation that can receive it as a rule annotated with @service and with a skip body:
tion, 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. To deal, instead, with camera failures and therefore support robustness, the Self-Healing Controller detects failures of other cameras based on a broadcast ping-echo mechanism.

To coordinate with other cameras, the managing subsystem can rely on the services provided by the component Communication Infrastructure. Precisely, to detect failures, the self-healing controller needs to coordinate with the self-healing controllers of other cameras using, respectively, the service interface ExtComService, to forward outgoing remote messages to the communication infrastructure, and PingEchoService to pass remote messages that arrive at the communication infrastructure to the local self-healing controller. Similarly, the organization controller needs to coordinate with organization controllers of other dependent cameras using the interfaces ExtComService and OrgService.

A component Knowledge is used to share and maintain information of the managed subsystem, the environment, and other relevant state for the MAPE computations. Actually, components interact directly via communication primitives and also indirectly via the knowledge. Note that this case study, as it is presented, does not require explicit adaptations of the architecture, but only changes to the knowledge repository. When a camera fails and remains silent, all dependent cameras will be aware of it and temporarily pause the relevant communications towards it without physically removing the connecting wires.

### 6.2 Self-adaptive components behavior

We specify the self-adaptive behavior of the overall distributed camera system using three main MAPE-K loops⁶:

\[
\{ \text{MAPE(flexibility), MAPE(intFailure), MAPE(extFailure)} \}
\]

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 of each camera. The second MAPE-K loop deals with the robustness concern to restructure organizations in case of failures of other cameras (silent cameras). It is handled by both the organization controllers and the self-healing controllers of each camera. Finally, the third MAPE-K loop deals with the robustness concern to restructure organizations in case of internal failure of the camera. The two managing agents of a camera are both involved in this loop.

Below, we describe the behavior of the organization controller by reporting some fragments of its SCA-ASM implementation. The complete SCA-ASM specification is available online [30].

**Organization controller.** A master/slave control model is adopted to structure organizations in case of congestion. To keep the master election policy simple, we assume every camera has a unique ID that is a monotonically increas-

---

**Listing 1: Program of each organization controller**

```
macro rule r_masterBevavior = 
    seq
      r_selfFailureAdapt[] //Adaptation due to internal failure
      r_failureAdapt[] //Adaptation due to external failure
      r_master_congestionAdapt[] //Adaptation due to congestion
    endseq

//Agent's program initialization
rule r_OrganizationController = r_masterBevavior[]
agent OrganizationController : r_organizationController[]
```

---

⁶Note that the concrete syntax `agent agent_type : rule[]` denotes in AsmetaL the initialization of the component agent’s function `program`.

---

3Note that the concrete syntax `agent agent_type : rule[]` denotes in AsmetaL the initialization of the component agent’s function `program`.

---

41
Listing 2: Excerpt of rule r_master_congestionAdapt

//@P
macro rule r_turnMasterWithSlaves =
  par
  r_send[knowledge(self),"addNewSlave(Camera)" ,cam(self)]
  r_send[cam(self),"getCameraState(CamState)" ,MASTERWITHSLAVES] program(self) = <<r_masterWithSlavesBehavior>>
endpar

//@P
macro rule r_turnSlave(smaster in Camera) =
  par
  r_send[knowledge(self),"getMaster","Cam(self),smaster"]
  r_removeSlavesTurningSlave[]
  program(self) = <<r_slaveTurningSlave>> //@E
  r_send[extOrg(self),"r_m_offer" ,($master, cam(self))]
endpar

//@P
macro rule r_send_s_offer =
  seq
    congested(self) := true
    r_send[extOrg(self),"r_m_offer" ,($master, cam(self))]
endseq

//@A
macro rule r_analyzeCongestion =
  if event(self)="m_offer" then
    r_turnMasterWithSlaves[]
  else
    if event(self)="s_offer" then
      seq
        r_sendreceive[knowledge(self),"getPrev" ,cam(self), prev(cam(self))]
        if isDef(prev(cam(self))) then
          r_turnSlave[prev(cam(self))]
        endif
      endseq
    endif
  endif
macro rule r_master_congestionAdapt =
  seq
    r_receive[requester(self),"r_event",event(self)]
    if isDef(event(self)) then
      if event(self)="no_cong" then //@M_s
        r_removeSlavesTurningSlave[]
      endif
    endif
endseq

Listing 3: Organization Controller's behavior in the roles slave and master with slaves

//@P
macro rule r_slave_congestionAdapt =
  seq
    r_receive[requester(self),"r_event",event(self)]
    if isDef(event(self)) then
      if event(self)="no_cong" then //@M_s
        r_removeSlavesTurningSlave[]
      endif
    endif
endseq

//@P
macro rule r_masterWithSlaves_congestionAdapt =
  seq
    r_receive[requester(self),"r_event",event(self)]
    if isDef(event(self)) then
      if event(self)="no_cong" then //@M_s
        r_removeSlavesTurningMaster[]
      endif
    endif
endseq

Note that atomic adaptation actions to restructure master/slave organizations (i.e., add a camera as slave of a master camera, set the new master of a slave camera, etc.) are services provided by the knowledge component (not reported here) and modeled as ASM rules annotated with @E.

The rules r_slave_behavior and r_masterWithSlaves_behavior for the roles SLAVE and MASTERWITHSLAVES, respectively, are defined similarly. They only differ in executing the rules r_slave_congestionAdapt and r_masterWithSlaves_congestionAdapt, respectively, as contribution to the third MAPE loop. These rules are reported in Listing 3. In the role of slave (rule r_slave_congestionAdapt in Listing 3), if the traffic is no longer jammed (event no_cong), the organization controller leaves the organization it belongs to by sending the event slaveGone to its master and becomes master of a single member organization by the rule r_turnMaster. Otherwise (still congested), the organization controller waits (by the rule r_receiveOrgSignals) for a restructuring event change_master or masterGone.

In the role of master with slaves (rule r_masterWithSlaves_congestionAdapt), when the traffic is no longer jammed, the event s_offer (see the plan rule r_send_s_offer in Listing 2) to the next alive camera (if any) in the direction of the traffic flow as request to join, as slave, the organization. Depending on the traffic condition of the next camera and its current role, the organizations may be restructured depending on the analysis computation represented by the rule r_analyzeCongestion (see Listing 2). If traffic is not jammed (the controlled predicate congested is false) in the viewing range of the next camera, organizations are not changed. Otherwise, organizations are joined:
This execution environment is based on the SCA through an execution platform for SCA-ASM [26]. Among different existing model validation approaches, in our work we have carried out simulation and from T2 to T3 for robustness (i.e., the ability of the system to adapt dynamically with changing conditions in the environment), and from T2 to T3 for robustness (i.e., the ability of the system to cope autonomously with errors during execution).

7. TMA VALIDATION

Model validation is a model analysis activity, less demanding than other analysis techniques as formal verification, to execute starting from the earlier stages of the model development. Validation is the process of investigating a model with respect to the user perceptions in order to ensure that the model really reflects the user needs and statements about the application, and usually permits to detect faults with limited effort.

Among different existing model validation approaches, in our work we have carried out simulation and scenario-based validation through an execution platform for SCA-ASM [26]. This execution environment is based on the SCA Tuscany runtime platform [32], and the ASM simulator AsmetaS [17] and validator AsmetaV [11] provided by the ASMETA framework [4]. Fig. 5 shows the validation process: the user can directly simulate an ASM-based specification in an interactive way (see Sect. 7.1) or write a scenario that automatizes the simulation and the checking of the produced output (see Sect. 7.2).

These two validation techniques allowed us to simulate different adaptation mechanisms described previously and be confident that the TMA model behaved as expected in the adaptation requirements. As better described in the next subsections, we simulated the TMA model with an increasing number of cameras and we reproduced different adaptation scenarios such as those shown in Fig. 2 from T0 to T2 for flexibility (i.e., the ability of the system to adapt dynamically with changing conditions in the environment), and from T2 to T3 for robustness (i.e., the ability of the system to cope autonomously with errors during execution).

7.1 Simulation

AsmetaS permits to perform either interactive simulation, where required inputs are provided interactively by the user during simulation, and random simulation, where inputs values are chosen randomly by the simulator itself. The simulator, at each step, performs consistent updates checking to check that all the updates are consistent: in an ASM, two updates are inconsistent if they update the same location to two different values at the same time [8]. In our case, by simulation we found (in a preliminary version of our specification) that a self-healing subsystem could, at the same time, turn the status of its camera both to MASTER and to FAILED (see the simulation trace in Fig. 6); that particular situation could occur when a camera c was already FAILED and the system received the events to both turn on and turn off the camera (i.e., both monitored locations startCam(c) and stopCam(c) were true). This consistency violation was due to a wrong scheduling of the operations of the self-healing subsystem.

Moreover, the AsmetaS simulator also permits to check if some invariants are satisfied during simulation. For example, we added to the specification the following invariants

\[
(state(c_1) \text{ FAILED} = state(c_{i-1}) \text{ FAILED}) \text{ implies next}(c_{i-1}) = c_{i+1} \\
(state(c_i) \text{ FAILED} = state(c_{i+1}) \text{ FAILED}) \text{ implies prev}(c_{i+1}) = c_{i-1}
\]

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 \(c_{i-1}\).

7.2 Scenario-based validation

Scenario-based validation is a more advanced way to simulate and inspect ASMs, by specifying a scenario representing a description of the actions of an external actor and the corresponding reactions of the system. There are two kinds of external actors:

- a user interacts with the system in a black box manner, by setting the values of the external environment (e.g., asking for a particular service), waiting for a step of the machine as reaction to his/her request, and checking the output values;
- an observer, instead, can also inspect the internal state of the system (i.e., values of machine functions) and check the validity of possible invariants of a certain scenario.

Scenarios are described in an algorithmic way using the text:
Listing 4: Flexibility validation scenario from T0 to T1 in Avalla

```asmeta
scenario Flexibility_T0_T1
load main.asm

set stopCam(c3) := false; set stopCam(c2) := false; set stopCam(c4) := false; set startCam(c3) := false; set startCam(c2) := false; set startCam(c4) := false; set startCam(c1) := false; set startCam(c1) := true; set slaves(c1,c2) := true; set slaves(c1,c3) := true; set slaves(c2,c3) := true; set slaves(c2,c4) := true; set slaves(c3,c4) := true; set state(c1) := MASTER; set state(c2) := MASTERWITHSLAVES; set state(c3) := SLAVE; set state(c4) := SLAVE;

exec par
  state(c2) := MASTERWITHSLAVES;
  state(c3) := SLAVE;
  state(c4) := SLAVE;
  getMaster(c1) := c1;
  getMaster(c2) := c2;
  getMaster(c3) := c3;
  getMaster(c4) := c4;
  congestion(c1) := false;
  congestion(c2) := true;
  congestion(c3) := false;
  congestion(c4) := false;
  congestion(organizationController4) := true;
  congestion(organizationController3) := true;
  congestion(organizationController2) := true;
endpar;

stop

set congestion(c2) := true;

check getMaster(c4) := c4 and kOffer(c2) = true and kOffer(c4) = false and slave(c3, c4) = true and state(c1) := MASTER and state(c2) := MASTERWITHSLAVES and state(c3) := SLAVE;

set congestion(c2) := false and newSlave(c3, c4) := true and state(c1) := MASTER and state(c2) := MASTERWITHSLAVES and state(c3) := SLAVE;

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

set congestion(c4) := false and newSlave(c2, c4) := true and state(c1) := MASTER and state(c2) := MASTERWITHSLAVES and state(c3) := SLAVE and state(c4) := SLAVE;
```

8. RELATED WORK

SA has been widely studied in the software architecture community [2]. Various mechanisms and frameworks for handling adaptation have been proposed, such as (to name a few): SA with aspect-orientation, Dynamic Reconfiguration, Model-Driven Development frameworks for SA, and frameworks for self-optimization (including the adaptation cost itself) [12, 14, 22, 25, 10, 24]. However, as shown in the study [34], little attention has been given to formal modeling and analysis of self-adaptive service-oriented applications. Here we review, in a non-exhaustive manner, the main contributions.

The work in [21] compares JOLIE, PiDuce and COWS, as candidate formalisms for modeling dynamically adaptable services. The analysis is, however, focused on time-constrained dynamic adaptation concerns. They illustrate strengths and limitations of each formalism from the point of view of the quality of service properties, such as availability and responsiveness, and justify the selection of COWS as the best-fit, though limited, language for studying compliance to time bounds.

The PLASTIC approach [6] uses the formal framework Chameleon to support context-aware adaptive services in Java applications. The approach is targeted to Java.

[23] concerns the formal specification and verification of dynamic reconfigurations of component-based systems using the B formal 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 of dynamic reconfiguration sequences to model-check. Their approach, though not conceived for service-oriented architectures, is similar to ours, but they support only architectural adaptation.

[18] uses architectural constraints specified in Alloy for the specification, design and implementation of self-adaptive architectures for distributed systems.

[15] proposes a model-based approach to combine self-adaptive composition and error recovery techniques to perform adaptation of BPEL or WF services. The authors present techniques to provide semi-automated support for identification and resolution of mismatches between service interfaces and protocols, and generate adaptation behavioural specifications.

[19] presents a model for dynamic reconfiguration of services using the service-oriented ADL SRMLight (a language derived from the more complex modeling language SRML developed within the EU project SENSORIA). Though they address only the specification of dynamic architectural characteristics of service-oriented applications, their mathematical model is general enough and useful to develop general assembly and binding mechanisms for service components in runtime middleware/platforms.

Timed Automata and Z are used in, respectively, [20] and [35], to model the Traffic Monitoring case study (the same presented here), and verify flexibility and robustness properties. 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 – most existing formal approaches to SA assume a centralized point of control –, and this is the reason why we were inspired mainly by them. However, Timed Automata and Z are used as they are without any extension for service-oriented applications, while SCA-ASM is a language for service-modeling. Moreover, the simplicity of the ASMs in expressing fundamental computing concepts allows one to avoid over-specification due to the rigidity of the formalism. That is, indeed, the main limitation of the approaches in [20, 35]. Such expressiveness makes, instead, the SCA-ASM method comprehensible to practitioners and feasible for large-scale applications. Similarly to our work, the goal-oriented framework Sim-
SOTA is proposed in [1] for modeling, simulating and validating MAPE-K feedback loop models of self-adaptive systems. However, SimSOTA adopts a semi-formal notation, namely UML activity diagrams, to model feedback loops.

Concerning lightweight modeling languages, in [19] an UML profile, called the control loop UML profile, is presented. It extends UML modeling concepts such that control loops become first class elements of the architecture. Another graphical notation to explicitly capture interacting MAPE loops is presented in [36]. The control loop UML profile supports modeling and reasoning about interactions between coarse-grained “controllers”, while the notation in [36] aims to model finer-grained interactions between the components of control loops and to define MAPE patterns (i.e., recurring structure of interacting MAPE components).

9. LESSONS LEARNED

Enhancing SCA-ASM with self-adaptation features was facilitated from the availability in SCA-ASM of modeling constructs for service-oriented applications, and of operators to model distributed computation and coordination/communication among components. By explicitly modeling MAPE-K control loops in SCA-ASM, we achieve a clear separation between adaptation logic and functional logic. This is possible since the formal approach allows us:

(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 separate, inside the behavior of a managing component, different adaptation concerns, each modeled as a separated transition rule (e.g., in the module OrganizationController, rule r_fallureAdapt[] models the robustness scenario, while rule r_master_congestAdapt[] models the flexibility scenario).

(iii) To keep separated, as invocations of separate rules, the four computations of a MAPE loop (e.g., rule r_master_congestAdapt[] corresponds to a monitoring computation, while rule r_analyzeCongestion[] is an analyzing computation, both rules of the same adaptation loop).

(iv) To distribute a given MAPE loop among different agents (e.g., the computation M to detect camera failures in case of robustness loop is in the SelfHealingController, while the other A,P,E are in the OrganizationController).

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 facilitates reasoning about component behavior and avoid over-specification, keeping models concise.

The availability of an execution platform for SCA-ASM, made possible to validate adaptation scenarios. Initially, a number of inconsistent updates arose, during the scenario simulation, due to the simultaneous execution of different adaptive behaviors performed by the same component. Validation activity helped us to reason about priorities of the different adaptation concerns and to fix the component models accordingly. The formalization of the different adaptive behaviors in terms of separate MAPE loop rules and a priority-based sequential scheme to coordinate their execution made it easy to set the correct component behavior thus avoiding unexpected interference among MAPE loops.

10. CONCLUSION AND FUTURE WORK

This work is part of our ongoing effort in defining a formal framework to model self-adaptive service-oriented applications, and to validate and verify abstract models in order to assure properties (e.g., flexibility, robustness, etc.) of self-adaptive applications already at design time.

By showing how to model self-adaptation in an SCA-ASM assembly of service components, we provide a framework for rigorous development of service-oriented applications exposing adaptive behavior in a decentralized manner and with partial shared knowledge. We applied this modeling approach to the Traffic Monitoring case study.

The mathematical rigor of the ASM formal method and the execution nature of ASM models make SCA-ASM not only a formal framework to develop design models, covering both structural and behavioral aspects, of self-adaptive service-oriented applications, but also a means to perform rigorous model analysis. In our work we have carried out model validation by means of the SCA-ASM simulation environment [30]. This model validation allowed us to reproduce the different adaptation behaviors as implemented by the managing components over the managed ones, and be confident that the model behaved as expected.

Model validation is a model analysis activity to be executed from the earlier stages of the model development before other more demanding but more sophisticated and complex analysis techniques such as formal verification. Currently, we are studying how to apply ASM model slicing techniques to tackle the verification of self adaptive SCA-ASMs and guarantee required properties (robustness, flexibility, safety, liveness, etc.) by model checking [3].

In the future, we plan to improve SCA-ASM to provide the designer with distributed coordination patterns for MAPE-K loop interactions.

11. ACKNOWLEDGMENTS

The work was partially supported by Charles University research funds PRVOUK.

12. REFERENCES


[29] OASIS/OSOA. Service Component Architecture


ABOUT THE AUTHORS:

Paolo Arcaini is assistant professor at the Faculty of Mathematics and Physics, Charles University in Prague (Czech Republic). He is member of the Formal Methods Group of the Department of Distributed and Dependable Systems. He graduated and received a PhD in Computer Science at the University of Milan, Italy. His research topics include specification, verification and runtime verification using Abstract State Machines, and model-based testing. He serves as member of the program committee of international conferences. He published papers in international journals and in proceedings of international conferences.

Elvinia Riccobene is associate professor in Computer Science at the Computer Science Department of the University of Milan, Italy. He teaches courses on Software Engineering and on System Modeling and Analysis. She is the director of the FALSE Lab (Formal Methods and Algorithms for Large Scale Systems Laboratory). She received master and PhD degree in Mathematics. Her research interests include formal methods, with particular expertise in the Abstract State Machines, integration between formal modelling and model-driven engineering, model-driven design of service-oriented applications and self-adaptive systems, and model analyses techniques for software systems. She has been the scientific coordinator of national research projects and she participated to various European and Italian research projects. She serves as member of the program committee of international conferences. She published several papers in international journals and in proceedings of international conferences.

Patrizia Scandurra is Assistant Professor at the Dep. of Management, Information and Production Engineering of the University of Bergamo (Italy). She teaches courses of software design/programming and operating systems. She received master and PhD degree in Computer Science. Her research interests include: integration of formal and semi-formal modelling languages; formal methods and functional analysis techniques; Software Architecture; Component-based Development; Service-oriented Computing; Model-driven Engineering; design and analysis techniques for distributed and self-adaptive systems, mobile/cloud systems, and embedded systems. She is a member of the Abstract State Machines (ASM) formal method community. She participated to several Italian and European research projects. She collaborates with companies and research centers. She serves as member of the program committee of international conferences. She published several papers in international journals and in proceedings of international conferences.