



MSL: A pattern language for engineering self-adaptive systems

Paolo Arcaini ^{a,*}, Raffaela Mirandola ^b, Elvinia Riccobene ^c, Patrizia Scandurra ^d



^a National Institute of Informatics, Japan

^b Politecnico di Milano, Italy

^c Università degli Studi di Milano, Dipartimento di Informatica, Italy

^d Università degli Studi di Bergamo, Dept. of Management, Information and Production Engineering, Italy

ARTICLE INFO

Article history:

Received 18 April 2019

Revised 9 December 2019

Accepted 20 February 2020

Available online 22 February 2020

Keywords:

Pattern-oriented modeling

Architecture-based self-adaptation

MAPE-K pattern loops

Self-adaptive ASMs

Adaptive smart home systems

ABSTRACT

In architecture-based self-adaptation of decentralized systems, *design patterns* have been introduced to ease the design of complex adaptation solutions that usually require the interaction of different MAPE-K (Monitor-Analyze-Plan-Execute over a shared Knowledge) control loops, each dealing with an adaptation concern of the managed system. Such MAPE patterns have been proposed by means of a graphical notation, but without a well-defined way to document them and to express the semantics of components interactions.

In this paper, we propose an approach to overcome these limitations. We present a domain-specific language, called *MSL* for *MAPE Specification Language*, to define and instantiate MAPE patterns and to give semantics to some semantic variation points of the equivalent graphical notation for MAPE pattern. We also provide a formal semantics of the language by means of *self-adaptive Abstract State Machines*, an extension of the Abstract State Machines (ASMs) formalism to model self-adaptation. Such semantics definition comes with an automatic transformation of MSL models into formal executable models, and opens to the possibility of performing rigorous analysis (validation w.r.t. the adaptation requirements and verification of adaptation properties) of MSL models. Moreover, we present our current results toward a (long-term) realization of an *MSL-centric framework*, where MSL is the notation of a modeling front-end, on top of richer and more specific modeling, analysis, and implementation back-end frameworks.

As proof of concept of our approach, we show the application of MSL and its formal support to a running case study in the field of home automation, by modeling an adaptive control of a virtual smart home developed with the OpenHAB runtime platform.

© 2020 Elsevier Inc. All rights reserved.

1. Introduction

Nowadays, software systems are expected to operate in uncertain and dynamic environments with highly changing operational conditions. *Self-adaptation* has been suggested as a viable and effective solution to deal with the increasing complexity, uncertainty and dynamics of these systems (Cheng et al., 2009; De Lemos et al., 2013; Kephart and Chess, 2003). In this area, feedback control loops that monitor and adapt managed parts of a software system are widely accepted as a key architectural solution (Brun et al., 2009) to realize self-adaptation in software systems. According to the original definition of Kephart and Chess (2003), we refer to these control schemas as MAPE-K (Monitor, Analyze, Plan, and Execute over a shared Knowledge) – or simply MAPE – feedback loops.

A single feedback loop is not enough to model the interactions of multiple concerns and aspects in large distributed systems, as outlined in Cheng et al. (2009). To this end, multiple interacting MAPE loops with a decentralized control are needed to deal with several adaptation concerns. To this purpose, common design patterns of interactive MAPE loops have been proposed in the literature (Weyns et al., 2013), together with a graphical notation for representing them. Instances of MAPE patterns capture the architectural view of the adaptation layer of self-adaptive systems, according to a precise design solution. Although graphical representation of patterns and their instances is able to express the structure of components interactions (who interacts with whom), it lacks of expressing their operational semantics (how and in which order). Fixing interaction semantics is necessary when moving from such diagrammatic representation to a formal specification to be used for property verification. Formal models of self-adaptive systems are, indeed, fundamental to guarantee reliability and correctness of the adaptation logic, both at design time and at runtime (Weyns et al., 2012). Moreover, although for

* Corresponding author.

E-mail address: arcaini@nii.ac.jp (P. Arcaini).

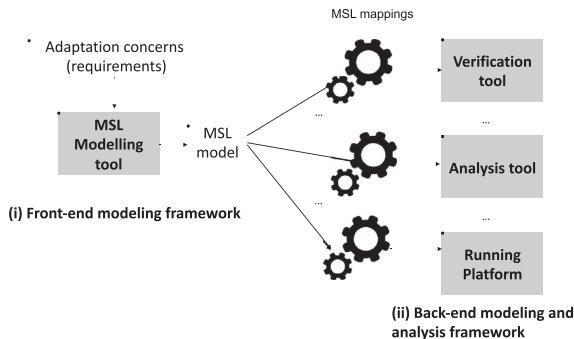


Fig. 1. The envisioned MSL-centric framework.

documentation purposes visual representation can facilitate comprehension, the use of a graphical notation usually negatively affects efficiency, effectiveness, and satisfaction of software developers when they perform analysis and modification tasks (Meliá et al., 2016).

To overcome these limitations, we propose a domain-specific language, called *MAPE Specification Language* (MSL), for architecting self-adaptive systems by using MAPE patterns and their instances as first-class citizens; in addition, the language has constructs to fix some semantic variations in MAPE component interactions. MSL adopts the same modeling concepts of the MAPE graphical notation presented in Weyns et al. (2013), but it uses a textual notation. The rationale in this decision is that textual notations should scale better than visual ones with increasing system design size (Meliá et al., 2016).

We also provide formal semantics to the MSL language in terms of the Abstract State Machine formal method (Börger and Raschke, 2018). Such semantic definition allows: (a) A(n automatic) transformation of MSL models of MAPE loops into formal executable models given in terms of *self-adaptive Abstract State Machines* (*self-adaptive ASMs*) (Arcaini et al., 2017), an extension of multi-agent ASMs to express self-adaptation; and (b) exploiting all the potentialities of a formal method to provide rigorous analysis (validation w.r.t. the adaptation requirements and verification of adaptation properties) of MSL models.

Besides the goal of having a notation able to overcome the above mentioned limitations regarding scalability and semantic ambiguousness of the notation in Weyns et al. (2013), our long-term envision is that of providing a rigorous, but user-friendly, framework (illustrated in Fig. 1) centered around the usage of MSL for the engineering of self-adapting systems. We view the MSL modeling front-end to be on top of richer and more specific modeling, analysis, and implementation back-end frameworks. Following this rationale, MSL provides a concise and simple notation to design fast architectural solutions in terms of MAPE patterns and their instantiations. More specific models or code in target back-end frameworks/platforms could be synthesized from these MSL models by using model transformations, and then tailored/refined according to the target scope.

The automatic mapping from an MSL model into a self-adaptive ASM provides the bridge toward the ASM-based formal back-end framework. By exploiting model transformations, different other back-ends (e.g., UML-like modeling notations, other ADLs, formal methods, simulation platforms or quality evaluation tools) can be integrated into the MSL-centric framework.

A smart home automation system is considered throughout the paper as running example to show the applicability of MSL and how to specify components interaction semantics. In particular, we adopted OpenHAB¹ as the running platform for implementing the

managed layer of a smart home whose adaptation/automation concerns have been modeled in MSL as instances of MAPE patterns. MSL models have been then refined at ASM level to capture the expected behavior of the MAPE components.

A preliminary version of MSL has been presented in the conference paper (Arcaini et al., 2018). With respect to Arcaini et al. (2018), in this paper we provide (i) an extended description of the MSL language by providing the modelling of MAPE patterns (some taken from the literature (Weyns et al., 2013) and other defined by the user), (ii) a precise semantics for MSL, (iii) an approach to re-design adaptation due to the results of a formal model analysis, (iv) a description of our (long-term) MSL-centric framework, and (v) a running case study in the field of home automation, by modeling an adaptive control of a virtual smart home developed with the OpenHAB runtime platform.

Paper organization. Some background on MAPE patterns for self-adaptation is given in Section 2. A presentation of the running case study is provided in Section 3. The MSL language is presented in Section 4 (including the language requirements and engineering approach, concrete and abstract syntaxes, parsing and validation), while its usage to model MAPE loops of the running case study is illustrated in Section 5. The MSL semantics in terms of a mapping from the MSL to the self-adaptive ASMs is described in Section 6. Some possible uses of an ASM-based framework, AS-META, for formal analysis of MAPE loops of the running case study are presented in Section 7. Section 8 describes how the outcome of the formal analysis phase can lead to a re-design of the self-adaptive architecture if some faults and/or loops interferences are discovered. A description of our MSL-centric framework is given in Section 9. Section 10 discusses the assessment of MSL with respect to the language requirements. Section 11 discusses possible threats that may affect the validity of the approach. Section 12 provides a description of some related work. Section 13 concludes the paper and outlines some future directions of our work.

2. Background on MAPE-patterns for self-adaptation

In architecture-based self-adaptation, the system is usually separated into the *managed subsystem* – the actual system (both hardware and software) that interacts with the environment and whose behavior can be changed according to the stimuli from the environment and the adaptation concerns – and the *managing subsystem* that includes the adaptation logic whose aim is to fulfill the adaptation concerns. This logic is typically conceived as a set of interacting MAPE loops, one per each adaptation concern. In Weyns et al. (2013), some recurring structures of interacting MAPE components, *MAPE patterns*, have been defined for designing decentralized adaptation solutions, where controllers make independent decisions but have some kind of interaction. Fig. 2 shows an example of such a pattern (the *Aggregate MAPE* pattern), an instance of this pattern, and the key symbols of the graphical notation adopted in Weyns et al. (2013).

A MAPE pattern defines the structure of a composite MAPE loop as a set of *abstract groups* of MAPE components representing the roles of the feedback processes and the *type of interactions* between MAPE components. A *pattern instance* describes the structure of the pattern for one particular configuration. The annotated multiplicity of the interactions between the groups of MAPE components determines the allowed occurrences of the different groups in the pattern. Referring to Fig. 2, we can distinguish different types of interactions:

- *Managing-managed subsystem interactions* are those between M components and the managed subsystem for monitoring purposes, and between E components and the managed subsystem for performing adaptations.

¹ <https://www.openhab.org>.

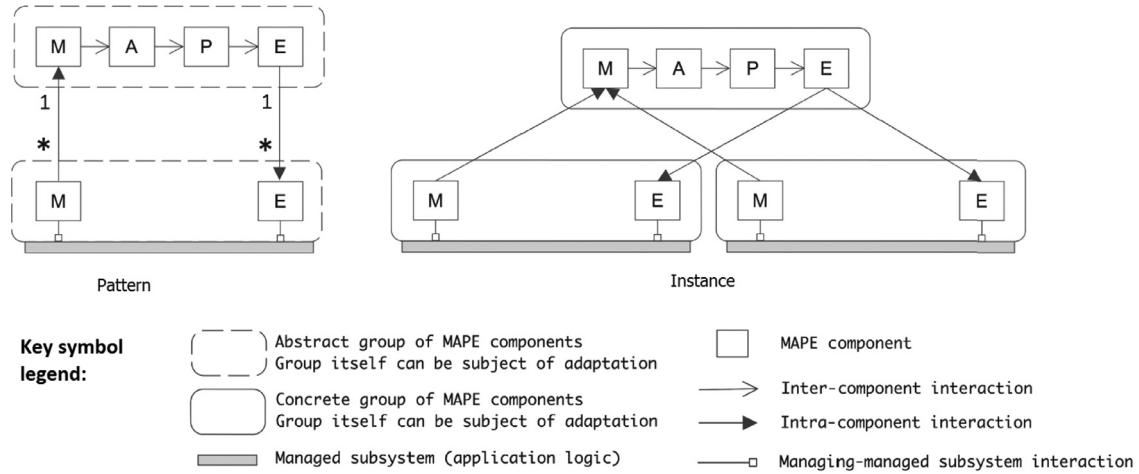


Fig. 2. Aggregate MAPE pattern (left) and an instance of it (right) from Weyns et al. (2013).

- *Inter-component interactions* are those between different types of MAPE components. Typically, M interacts with A, A with P, and P with E.
- *Intra-component interactions* are those between MAPE components of the same type, e.g., interactions between M components.

Although it is undoubtedly the importance of MAPE patterns to represent known design solutions and support their reuse, the semantics of their graphical representation is often ambiguous and may intentionally leave *semantic variation points* (as in UML). Given the MAPE pattern in Fig. 2, elements of ambiguous interpretation are, for example, the AND/OR semantics of signals when an M computation of the higher MAPE group is triggered by the M computations of the lower groups. In MSL, we allow the specification of such semantic variations and support the designer in fixing the semantics at configuration level.

3. Running case study: Adaptive control of a smart home

Smart home automation systems have been around for several years now (Darby, 2018). However, designing them in a way that user well-being is guaranteed in every situation is a very complex task. Typical real-life situations, indeed, involve conflicts that can arise in the home environment because, for example, unexpected events happen and/or because the home occupants are simultaneously engaged in different activities with different and discordant preferences for the home appliances.

Dealing with these situations require mechanisms that are able to handle unexpected and uncertain operational conditions. To this end, self-adaptive systems can offer the desired solution architecting the software control system of a smart environment with two layers: the managed layer, for basic control operations, and a managing layer realizing self-adaptation mechanisms for more advanced control operations/processes.

In this case study, the managed system has been modeled using the *Open Home Automation Bus* (OpenHAB),² which is an open source software platform for home automation. Being hardware/protocol agnostic, OpenHAB allows users to integrate and connect a variety of devices from classical home automation systems to new IoT gadgets and devices. Device (functionalities) are represented in terms of *items*, which have a *state* that can be changed by *commands*. A lightweight user-friendly dashboard interface for OpenHAB, the HABPanel, may be set up and made accessible from a standard web browser or smartphone/tablet. Through

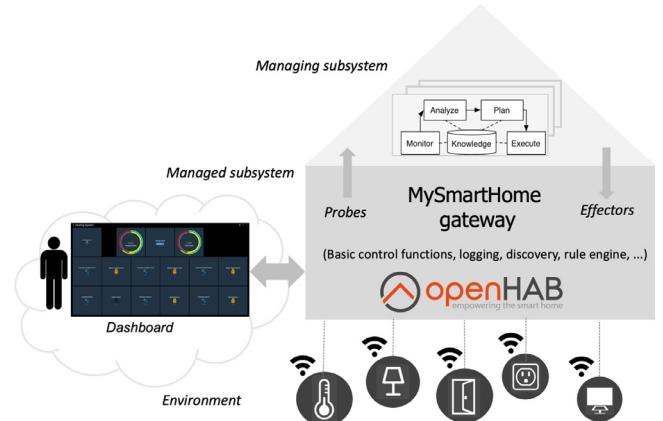


Fig. 3. Self-adaptive architecture of the smart home control system.

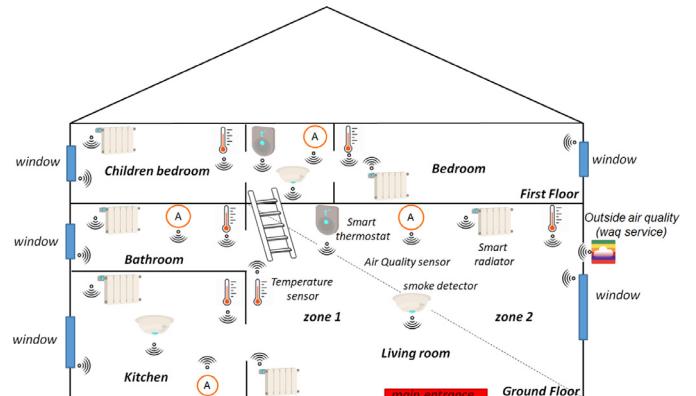


Fig. 4. (Virtual) smart home example.

this dashboard, the user can observe values of the probes and the effectors of the smart home devices.

The managing system is conceived as a set of MAPE-K feedback loops to deal with several concerns and make the smart home gateway more robust. The structural architecture of our (virtual) smart home is illustrated in Fig. 3.

We configured the OpenHAB instance for a virtual smart home³ as depicted in Fig. 4. The home is endowed with the devices necessary to run the automation processes presented in this article. The smart home is composed of two floors. Each floor has a smart

² <https://www.openhab.org/>.

³ See the GitHub project <https://github.com/scandurra/VirtualSmartHome-Project>.

thermostat communicating with different temperature sensors distributed among the rooms of the floor and with smart thermostatic valves (“smart radiator” in the figure) of the rooms heating of the floor. Air quality sensors, window opening control devices, and smoke/flame detectors are also present and configured as Thing’s items of interest in the OpenHAB platform.

Adaptation concerns. To cope with the design complexity, we separate the adaptation logic from the basic control system functionality and structure it in terms of MAPE-K feedback loops according to the following adaptation concerns. Note this list of concerns is only representative and used with the purpose of illustrating the MSL language.

Heating comfort (HC) At any time of the day, the system uses several room temperature sensors to achieve a high accuracy in the measures and regulates the heating at sufficient speed setting to ensure a comfortable temperature accordingly. Moreover, in order to minimize dispersion, the system holds windows closed when the heating is on.

Fire detection (FD) The system should detect and respond to the presence of a flame or a fire, allowing fire detection continuously.

Air quality (AQ) Air quality inside the home is held at good level by opening/closing windows according to the measures provided by indoor air quality sensors and by an external web service for outdoor air quality.

These concerns might lead to (conflicting) sub-concerns and are also cross-cutting since they work on shared items and their implementation is distributed across multiple system modules. For this reason, they require modularization techniques and design patterns that define how they should relate to each other.

4. MSL: A DSL for MAPE patterns specification

MSL is a textual DSL for modeling the adaptation layer of a self-adaptive system.

The following subsections overview the main language requirements and development approach (Section 4.1), the MSL modeling constructs and concrete syntax (Section 4.2), and the MSL abstract syntax including parsing/validation aspects (Section 4.3). The MSL semantics is instead defined in Section 6 in terms of self-adaptive ASMs, while the status of the tools implementation around MSL is presented in Section 9.

4.1. MSL language requirements and engineering approach

To design the MSL language, we tried to adhere to proven design principles in software architecture (Microsoft, 2009) and DSL engineering (Voelter et al., 2013) and, particularly, we identified the following set of requirements as key design dimensions of the language:

- *Model purpose:* The DSL is expected to be used as a modeling front-end to provide a concise and simple model of the composite structure of interactive MAPE-K loops, from which (for example, using model transformations/generators) a corresponding representation in a target back-end framework can be obtained and then tailored/refined according to the target scope (code implementation of MAPE components, formal behavior specification and analysis of MAPE-K loops, etc.), thus providing additional standalone independent models/code. The primary purpose of the model is therefore to serve as shared understanding of the MAPE-K loops structure between different solution spaces and associated frameworks/software platforms plugged-in the DSL-centric framework as back-end modules.

- *Separation of concerns:* Introduce language abstractions that allow dividing the adaptation logic of a self-adaptive software system into distinct adaptation concerns by structuring it into different MAPE loops with as little overlap in functionality as possible.
- *Principle of Least Knowledge (or Law of Demeter):* A MAPE loop/component should not know about internal details of other MAPE loops/components. The advantage of following the Law of Demeter is that the resulting software tends to be more maintainable and adaptable.
- *Minimize upfront design:* Introduce language constructs allowing designing what is necessary, thus to avoid making a large design effort prematurely.
- *Pattern-oriented design:* Combine individual patterns of MAPE loops into heterogeneous compound structure of MAPE loops to facilitate a constructive development of the system’s adaptation layer.
- *Model composability:* At a certain point of the design, models of MAPE-K loops obtained by separation of the adaptation concerns demand for model composition. To this end, a fundamental problem is assuring the ability to compose interfering MAPE-K loops to form dependable managing sub-systems without violating the desired adaptation requirements, and without diminishing the resulting trustworthiness.

Section 10 discusses the assessment of MSL with respect to these requirements.

Regarding the language engineering approach, we adopted the same concepts of the graphical notation introduced in Section 2 as core concepts of MSL for defining MAPE patterns and their instances. Additionally, further language constructs were added for modeling semantic variation points about MAPE component interactions explicitly. The MSL textual notation (the concrete syntax) has been developed using the grammarware approach of Xtext combined with the modelware approach of the Eclipse Modeling Framework (EMF); this allows the automatic generation of a model editor, a parser, and a basic validator. The Xtext grammar of the MSL notation together with the modeling/code artifacts of the running case study used in this article are available online at [The MSL language \(2018\)](#).

4.2. MSL modeling notation

Pattern definition. Core modeling elements of MSL to define a MAPE pattern in terms of abstract groups of MAPE components, managed subsystems (i.e., a set of observable items/parts of a managed systems), and their interactions with multiplicities are reported in Fig. 5. For each element, the corresponding graphical notation adopted in Weyns et al. (2013) and the textual notation in MSL are shown. The MSL syntax is intuitive and self-explanatory.⁴ Note that the MSL notation allows to be more specific than the graphical notation on some aspects; for example, the graphical notation used in Weyns et al. (2013) does not allow to distinguish between a single managed system and one composed of different subsystems, while this is possible in MSL: This is why the graphical notations of rows 2 and 3, and 4 and 5, are the same.

A MAPE pattern is defined by introducing a named element *abstract pattern* (by the keyword `abstract pattern`) that declares the managed subsystems’ type, the abstract groups of MAPE components, and the type of interactions between MAPE components. Code 1 reports the MSL definition of a simple MAPE pattern (a single group of MAPE components), and Code 2 reports

⁴ Note that we do not provide keywords to distinguish between intra- and inter-interactions, since they are already characterized by the kind of MAPE components connected by the interaction.

MAPE element	Graphical notation	MSL notation
Abstract group AGrp of MAPE components		group AGrp { components M, A, P, E }
Managed system Sys		system Sys
Managed system divided into subsystems Sys1, Sys2, etc.		system Sys1 system Sys2 ...
Managing-managed interaction between an abstract group AGrp and a system Sys		system Sys group AGrp { managedSys Sys components M, A, P, E }
Managing-managed interaction between an abstract group AGrp and subsystems Sys1, Sys2, etc.		system Sys1 system Sys2 ... group AGrp { managedSys Sys1,Sys2,... components M, A, P, E }
Inter-component interaction in an abstract group AGrp		interaction AGrp.X -> AGrp.Y [m1,m2]
Inter-component interaction between two abstract groups AGrp1 and AGrp2		interaction AGrp1.X -> AGrp2.Y [m1,m2]
Intra-component interaction in an abstract group AGrp		interaction AGrp.X -> AGrp.X [m1,m2]
Intra-component interaction between two abstract groups AGrp1 and AGrp2		interaction AGrp1.X -> AGrp2.X [m1,m2]
Managing-managed interaction between an abstract group AGrp0 and another abstract group AGrp1		group AGrp1 { components M, A, P, E } group AGrp0 { managedGrp AGrp1 [m] components M, A, P, E }
Managing-managed interaction between an abstract group AGrp0 and other abstract groups AGrp1, AGrp2, etc.		system Sys group AGrp1 { components M, A, P, E } group AGrp2 { managedSys Sys components M, A, P, E } ... group AGrp0 { managedGrp AGrp1[m1], AGrp2[m2], ... components M, A, P, E }

Fig. 5. MAPE pattern definition.

```
abstract pattern SimpleMAPE {
  system Sys

  group Main {
    managedSys Sys
    components M, A, P, E
  }

  interaction Main.M -> Main.A
  interaction Main.A -> Main.P
  interaction Main.P -> Main.E
}
```

Code 1. Simple MAPE definition in MSL.

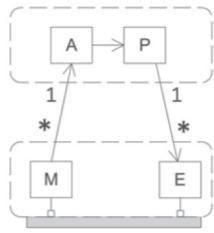
```
abstract pattern AggregateMAPE {
  system Sys
  group Main {
    components M, A, P, E
  }
  group Interface {
    managedSys Sys
    components M, E
  }
  interaction Interface.M -> Main.M [*--SOME,1]
  interaction Main.E -> Interface.E [1,*--SOME]
  interaction Main.M -> Main.A [1,1]
  interaction Main.A -> Main.P [1,1]
  interaction Main.P -> Main.E [1,1]
}
```

Code 2. Aggregate MAPE pattern definition in MSL.

Table 1

Description of the MAPE patterns proposed in Weyns et al. (2013).

Pattern	Description
Aggregate	The whole adaptation decision is performed by a single centralized MAPE loop whose M component aggregates the information coming from different distributed M components, and the E component communicates with distributed E components to perform the adaptation.
Coordinated control	The four MAPE activities are decentralized over different loops. Each component of each loop coordinates with corresponding components in other loops.
Information sharing	It is similar to the coordinated control, but only M components of different loops interact with each other. The other components only communicate inside their loop.
Master-slave	The analysis and planning (A and P components) are performed in a single master loop, while monitoring and execution (M and E components) are distributed among different slave loops that communicate with the central master loop.
Regional planner	A regional planner is constituted of a loop performing the planning (P component) communicating with other local loops performing the other activities (M, A, and E components); a regional planner performs adaptation in the local region. Different regional planners can communicate through the P component of their region to coordinate adaptations that span different regions.
Hierarchical control	Different simple MAPE loops are provided for different concerns at different levels of abstraction. The MAPE loops are structured hierarchically, where the MAPE loops at the bottom interact with the managed subsystem, while MAPE loops at intermediate layers interact with the adaptation layers beneath. The top MAPE loop overviews the overall adaptation.



```
abstract pattern MasterSlaveMAPE {
    system Sys

    group Master {
        components A, P
    }

    group Slave {
        managedSys Sys
        components M, E
    }

    interaction Slave.M -> Master.A [*--SOME,1]
    interaction Master.P -> Slave.E [1,*--SOME]
    interaction Master.A -> Master.P [1,1]
}
```

Code 3. Master-slave pattern definition in MSL.

the MSL definition of the Aggregate MAPE pattern shown in Fig. 2. Code 3 shows the MSL definition of the Master-Slave MAPE pattern from Weyns et al. (2013) (on the top) and its corresponding MSL definition (on the bottom). All patterns proposed in Weyns et al. (2013) can be expressed in MSL and are available online (see the pattern library at [The MSL language \(2018\)](#)). Their description is briefly reported in Table 1.

Another (more complex) example of pattern definition is reported in Code 4. This pattern captures a hierarchical distribution control, where a higher-level MAPE group manages subordinate MAPE groups (i.e., the managed subsystems can be managing subsystems themselves). Specifically, it is an example of composition by hierarchy of known abstract patterns, aggregate MAPE and simple MAPE, that lead to a new abstract pattern. This abstract pattern will be used in Section 8 to compose and solve the two interfering concerns related to heating comfort and air quality, respectively, of the running case study.

Semantic variation points at interaction level. One of the ambiguities of the MAPE loop representation proposed in Weyns et al. (2013) is related to the interpretation of the interactions [1,*], [*,1], and [*,*] among multiple components of different MAPE groups. Indeed, it is not clear whether, in order to trigger the interaction,

```
abstract pattern HierarchicalControlMAPE {
    system SysOne
    system SysTwo

    //Interface group for Aggregate MAPE
    group InterfaceOne {
        managedSys SysOne, SysTwo
        components M, E
    }

    //Group for a Simple MAPE
    group InterfaceTwo {
        managedSys SysTwo
        components M, A, P, E
    }

    //Main group for Aggregate MAPE
    group Intermediate {
        components M, A, P, E
    }

    //Higher group of the hierarchy
    group High {
        managedGrp Intermediate [*--ALL],
        InterfaceTwo [*--ALL]
        components A, E
    }

    interaction InterfaceOne.M -> Intermediate.M [*--SOME,1]
    interaction Intermediate.E -> InterfaceOne.E [1,*--SOME]
    interaction Intermediate.M -> Intermediate.A [1,1]
    interaction Intermediate.A -> Intermediate.P [1,1]
    interaction Intermediate.P -> Intermediate.E [1,1]
    interaction InterfaceTwo.M -> InterfaceTwo.A [1,1]
    interaction InterfaceTwo.A -> InterfaceTwo.P [1,1]
    interaction InterfaceTwo.P -> InterfaceTwo.E [1,1]
    interaction High.A -> High.E [1,1]
}
```

Code 4. A hierarchical control MAPE pattern definition in MSL.

the communication must be established among all the involved components or only some of them. Therefore, in MSL, in addition to the standard multiplicity 1, we allow the specification of the intended semantics of * by means of multiplicities *-ALL, *-SOME, and *-ONE (see, for example, interactions in Codes 2,3, and 4). When used as source multiplicity of the interaction, these multiplicities respectively mean that the target group must receive the communication from *all* the interacting groups, from *at least one* of them, or from *exactly one* of them. In a similar way, when used as target multiplicity of the interaction, they mean that the source group must communicate with *all* the interacting target groups, a *non-empty subset* of them, or *exactly one* of them.⁵ Note that in

⁵ If no multiplicity is reported, [1,1] is used as default.

```

import AggregateMAPE
concrete pattern HC_MAPE
concretizationOf AggregateMAPE {
    system Heating: AggregateMAPE.Sys
    group MainHC: AggregateMAPE.Main
    group IntHC: AggregateMAPE.Interface
}
configuration MySmartHomeHC instanceOf HC_MAPE {
    hs_ff: HC_MAPE.Heating //heating system of First Floor
    hs_gf: HC_MAPE.Heating //heating system of Ground Floor
    //main group for the concern Heating Comfort
    main_hc: HC_MAPE.MainHC {
        components m_hc:M, a_hc:A, p_hc:P, e_hc:E
    }
    //interface group for the ground floor
    int_hc_gf: HC_MAPE.IntHC {
        managedSys hs_gf
        components m_hc:M, e_hc:E
    }
    //interface group for the first floor
    int_hc_ff: HC_MAPE.IntHC {
        managedSys hs_ff
        components m_hc:M, e_hc:E
    }
    //intra-interactions
    int_hc_gf.m_hc -> main_hc.m_hc
    int_hc_ff.m_hc -> main_hc.m_hc
    main_hc.e_hc -> int_hc_gf.e_hc
    main_hc.e_hc -> int_hc_ff.e_hc
    //inter-interactions
    main_hc.m_hc -> main_hc.a_hc
    main_hc.a_hc -> main_hc.p_hc
    main_hc.p_hc -> main_hc.e_hc
}

```

Code 5. MAPE loop for the Heating Comfort concern.

case of *-ALL, the MSL semantics does not specify *when* the different communications must be sent (in case it is used as target multiplicity) or received (in case it is used as source multiplicity); such *temporal semantics* will be provided by the corresponding ASM model. Similarly, also in case of *-SOME, and *-ONE, the MSL semantics does not specify *which ones*: the implementation of the *choice policy* will be specified in the ASM model.

Pattern instantiation. Once defined, an abstract MAPE pattern can be instantiated by the designer in an MSL model of her/his control loop architecture for a certain adaptation concern.

Code 5 shows the MSL model of the MAPE feedback loop for the concern Heating Comfort of the running case study. It is an instance, by the `import` clause,⁶ of the aggregate MAPE pattern. A MAPE pattern instantiation consists of two phases: (i) *Concretization* and (ii) *configuration*.

First, a *concrete pattern* (a named element preceded by the keyword `concrete pattern`) must be introduced as *concretization* of the abstract pattern to tailor the abstract pattern to a specific application domain. In this way, we ensure the structure of the pattern is the same, but we allow role renaming (i.e., the names of the abstract MAPE groups and managed subsystems) of the abstract pattern to reflect better the roles of the specific domain, and therefore the intended usage/reuse. Roles renaming is realized by *name binding*, i.e., through identifiers. The first part of Code 5 shows a concretization, called HC_MAPE, of the pattern AggregateMAPE for the concern Heating Comfort of the running case study.⁷ Essentially, there is a concrete MAPE group, IntHC,

that is responsible for monitoring rooms temperature via sensors and manages the heating subsystem (the managed subsystem) accordingly. So it plays the role of *Interface* w.r.t. the heating subsystem by providing both a component M and a component E. The group MainHC is responsible for realizing the adaptation concern; therefore, its component M aggregates temperature data from all temperature sensors through the components M of IntHC, and then its component A decides if it is necessary to adjust or not the heating settings. In case adaptation is required, components P and then E are triggered to plan adaptation actions and drive the components E of IntHC, respectively.

A concrete pattern is still at type-level, so to bring it at instance-level it has to be instantiated for a specific scenario. To this purpose, a *configuration* (a named element introduced by the keyword `configuration`) must be introduced in the MSL model. Such a configuration instantiates the concrete groups of MAPE components and managed subsystems that effectively play the roles (as renamed) of the concrete pattern and their (concrete) interactions. The second part of Code 5 (as introduced by the keyword `configuration`) shows a configuration, called MySmartHomeHC, of the concrete pattern HC_MAPE for the smart home depicted in Fig. 4. In this configuration, there are two managed heating subsystems (hs_gf and hs_ff) for the two floors, Ground Floor (GF) and First Floor (FF), of the home, and their interface MAPE groups instances (int_hc_gf and int_hc_ff) interacting with one main MAPE group (main_hc).

Other examples of MSL models of MAPE feedback loops are reported in Section 5 to complete the specification of all the adaptation concerns of the running case study.

4.3. MSL abstract syntax, parsing, and validation

In Xtext, an EMF Ecore model (an object graph) is inferred automatically from the grammar of a language and then used as the in-memory representation of any parsed text file. Depending on the community, this in-memory object graph is called the abstract syntax tree (AST) or document object graph (DOM) or model instance of a metamodel (the classes typing the objects and their relationships). We use here the term (meta)model and AST interchangeably to denote the MSL abstract syntax that abstracts over syntactical information. The MSL metamodel as class diagram of Ecore EClasses is available online at [The MSL language \(2018\)](#).

The MSL metamodel is used by later processing steps in Xtext, such as validation, compilation or interpretation or artifacts/code generation. The MSL parser generated automatically by Xtext has been complemented, for example, by a validator to perform static analysis of MSL models and give informative feedback to the users. In particular, in addition to standard Xtext validation⁸, we specified some constraints for pattern matching, i.e., to check whether a given pattern instance (configuration of a concrete pattern) conforms to its abstract pattern. We implemented such constraints using Xtend into the Xtext validator. The OCL formalization of two of these constraints is as follows:

- *Checking concrete systems in a concrete pattern w.r.t. the abstract pattern:* each concrete pattern must declare one concrete system per abstract system.

⁶ We allow the definition and instantiation of a MAPE pattern in the same MSL file with extension .msl. In order to create a library of patterns, we also allow the definition of patterns without instantiation, that can be later imported.

⁷ All the developed MSL models and generated ASM models are available online at <https://github.com/fmselab/msl/tree/master/examples/SmartHomeGateway/>.

⁸ Xtext provides *syntactical validation* with the parser (to detect syntax errors), *cross-reference validation* with the linker (to detect broken links), and *concrete syntax validation* with the serializer (e.g., for type checking). Please refer to the Xtext documentation for more details: https://www.eclipse.org/Xtext/documentation/303_runtime_concepts.html#validation.

```

context Pattern
def: ap: AbstractPattern = self.absPattern
inv: ap.managedSystems?->forAll(
    as:AbstractSystem |
    self.managedSystems-->
    select(s:SystemBinding |
    s.absSys = as.name)-->size() = 1)

• Checking concrete groups in a concrete pattern w.r.t. the abstract pattern: each concrete pattern must declare one concrete group per abstract group.

context Pattern
def: ap: AbstractPattern = self.absPattern
inv: ap.abstractGroups?->forAll(
    ag:AbstractGroup |
    self.groups-->select(g:GroupBinding |
    g.absGroup = ag.name)-->size() = 1)

```

More complex constraints check that each interaction defined in the configuration must match in terms of source and target groups and source and target components, with the interactions defined in the abstract and concrete patterns.

5. MSL design of a smart home system

In order to model in MSL the adaptive control of our smart home environment, we first considered the adaptation concerns mentioned previously in [Section 3](#) and organized them (see [Table 2](#)) by type of goal that adaptation would meet, by sources of uncertainty (that make adaptation necessary), and type(s) of adaptation to perform dynamically.

Below, we describe each concern and associated adaptation goal, and report the MSL models of the MAPE-K feedback loops that deal with these concerns.

5.1. Heating Comfort (HC)

For the concern HC, we aim at ensuring *heating comfort* by introducing a finer-grained control feature that adjusts the heating system at sufficient settings to allow regulation of radiators' valves (and, therefore, of the pump speed) depending on the real warm necessity accounted by the multiple room sensors, thus avoiding over- or under- heating. Moreover, in order to avoid warm dispersion, a hold-closed feature is added that keeps windows closed when heating is on.

The MSL MAPE feedback loop for this concern has been already presented in the previous [Section 4](#) (see [Code 5](#)). We here refine it to separate the managed windows subsystem from the managed heating subsystem. To this purpose, the definition of the Aggregate MAPE pattern is also revised to allow for the specification in the concrete pattern of two types of managed subsystems (Heating and Air) denoting two different sets of managed items (heating and windows). [Code 6](#) reports this extended MAPE loop pattern definition and its instantiation.

Table 2

Adaptation requirements.

Concern	Adaptation goal	Source of uncertainty	Type of adaptation
HC	Guarantee heating comfort	Variable heat necessity Some windows may be accidentally open	Adjust heating at sufficient settings Hold windows closed when heating is on
FD	Guarantee fire detection sensing	Failure of fire/smoke detection sensors	Sensing functionality by exploiting other devices (e.g., heating sensors)
AQ	Guarantee air quality	Unpredictable air pollution levels	Hold windows closed/open

```

abstract pattern AggregateMAPEplus {
    system SysOne
    system SysTwo

    group Main {
        components M, A, P, E
    }

    group Interface {
        managedSys SysOne,SysTwo
        components M, E
    }

    interaction Interface.M -> Main.M [*--SOME,1]
    interaction Main.E -> Interface.E [1,*--SOME]
    interaction Main.M -> Main.A [1,1]
    interaction Main.A -> Main.P [1,1]
    interaction Main.P -> Main.E [1,1]
}

concrete pattern HC_MAPE
concretizationOf AggregateMAPEplus {
    system Heating: AggregateMAPEplus.SysOne //Heating sys
    system Air: AggregateMAPEplus.SysTwo//Windows sys
    group MainHC: AggregateMAPEplus.Main
    group IntHC: AggregateMAPEplus.Interface
}

configuration MySmartHomeHCplus instanceOf HC_MAPE {
    hs_ff: HC_MAPE.Heating //heating system of First Floor
    hs_gf: HC_MAPE.Heating //heating system of Ground Floor
    ws_gf: HC_MAPE.Air //windows system of Ground Floor
    ws_ff: HC_MAPE.Air //windows system of First Floor

    //main group for the concern Heating Comfort
    main_hc: HC_MAPE.MainHC {
        components m_hc:M, a_hc:A, p_hc:P, e_hc:E
    }

    //interface group for the ground floor
    int_hc_gf: HC_MAPE.IntHC {
        managedSys hs_gf, ws_gf
        components m_hc:M, e_hc:E
    }

    //interface group for the first floor
    int_hc_ff: HC_MAPE.IntHC {
        managedSys hs_ff, ws_ff
        components m_hc:M, e_hc:E
    }

    //inter-interactions
    main_hc.m_hc -> main_hc.a_hc
    main_hc.a_hc -> main_hc.p_hc
    main_hc.p_hc -> main_hc.e_hc
    //intra-interactions
    int_hc_gf.m_hc -> main_hc.m_hc
    int_hc_ff.m_hc -> main_hc.m_hc
    main_hc.e_hc -> int_hc_gf.e_hc
    main_hc.e_hc -> int_hc_ff.e_hc
}

```

Code 6. MAPE loop for the concern HC (extended).

Note that MSL allows only to define the MAPE components and connectors constituting the control architecture, while the behavior of the modeled MAPE components (how to monitor, how to analyze the knowledge, etc.) should be specified by the developer using the implementation language of a target back-end platform integrated within the MSL framework. In [Sections 6](#) and [7](#), we describe, for example, how to formally specify the behavior of the MAPE components using the formalism of self-adaptive ASMs.

5.2. Fire Detection (FD)

With this concern FD, we aim at providing a more robust safety feature by guaranteeing *fire detection* sensing. Being a domestic environment, usually flame/smoke sensors detect the presence of fire or flames. If smoke/flame is detected by the sensors, the gateway notifies the user through an alarm and/or sends a message to the user smartphone. To guarantee fire detection, the managing system triggers the alarm even if none of the flame/smoke detectors switches to ON (e.g., they may be silent because of a failure) but a room temperature sensor reports a temperature greater than a certain threshold (e.g., 45°C). This policy is an example of re-allocation of functionality on differently capable devices, typically used in IoT scenarios to allow architecting *emergent configurations* (Alkhabbas et al., 2017) when some devices may be silent (or failed). In our case, the fire sensing operation of a failed/silent fire/smoke sensor is temporarily realized by the temperature sensors in the home.

[Code 7](#) reports the MAPE loop pattern for the concern FD as instantiation of the Master-Slave MAPE pattern. Being a domestic building, fire detection takes the form of a fire alarm system, incorporating three smoke devices in two zones (the two managed subsystems fd_gf and fd_ff): two in the kitchen and living room (ground floor), and one in the first floor between the two bedrooms. Two slave components gf_slave and ff_slave are responsible for monitoring rooms temperature and execution of alarms activation in the corresponding floors, while a (centralized) master component is responsible for the analysis and planning of proper actuation.

5.3. Air Quality (AQ)

With this concern AQ, we aim at guaranteeing a good *air quality* inside the home by providing a hold- open/close feature for windows to restore good air values depending on the indoor

```
import MasterSlaveMAPE
concrete pattern FD_MAPE
concretizationOf MasterSlaveMAPE {
    system FireDetection: MasterSlaveMAPE.Sys
    group MasterFD: MasterSlaveMAPE.Master
    group SlaveFD: MasterSlaveMAPE.Slave
}

configuration MySmartHomeFD instanceOf FD_MAPE {
    fd_ff: FD_MAPE.FireDetection // for First Floor
    fd_gf: FD_MAPE.FireDetection // for Ground Floor

    //master
    fd.master: FD_MAPE.MasterFD {
        components a_fd:A, p_fd:P
    }

    //ground floor slave
    gf_slave: FD_MAPE.SlaveFD {
        managedSys fd_gf
        components m_gf:M, e_gf:E
    }

    //first floor slave
    ff_slave: FD_MAPE.SlaveFD {
        managedSys fd_ff
        components m_ff:M, e_ff:E
    }

    //interactions
    fd.master.a_fd -> fd_master.p_fd
    gf_slave.m_gf -> fd_master.a_fd
    ff_slave.m_ff -> fd_master.a_fd
    fd_master.p_fd -> gf_slave.e_gf
    fd_master.p_fd -> ff_slave.e_ff
}
```

[Code 7](#). MAPE loop for the concern FD.

```
import SimpleMAPE
concrete pattern AQ_MAPE
concretizationOf SimpleMAPE {
    system Air: SimpleMAPE.Sys
    group MainAQ: SimpleMAPE.Main
}

configuration MySmartHomeAQ instanceOf AQ_MAPE {
    aqs_gf: AQ_MAPE.Air //air system of Ground Floor
    aqs_ff: AQ_MAPE.Air //air system of First Floor

    aq_main: AQ_MAPE.MainAQ {
        managedSys aqs_gf, aqs_ff
        components m_aq:M, a_aq:A, p_aq:P, e_aq:E
    }

    //interactions
    aq_main.m_aq -> aq_main.a_aq
    aq_main.a_aq -> aq_main.p_aq
    aq_main.p_aq -> aq_main.e_aq
}
```

[Code 8](#). MAPE loop for the concern AQ.

air sensors' measures. We assume an internal air quality index as measured by common indoor air sensors by aggregating different air parameters (CO_2 , NO_2 , CO, etc.) and evaluate it according to the classification levels of the epa.gov agency. We adopted the web service www.aqicn.org to obtain an air quality index for the air outside. This index is then evaluated according to the classification levels of the ARPA agency in Italy.⁹

[Code 8](#) reports the MAPE loop pattern for the concern AQ as instantiation of the simple MAPE pattern (a single group of MAPE components, as shown in [Code 1](#)). The air subsystem of a floor denotes a set of indoor air sensors, the windows system, and the outdoor air web service.

6. The MSL semantics

To provide formal semantics to the MSL language (whose abstract syntax can be defined in terms of a metamodel, that inferred by Xtext), we here exploit the *semantic mapping* technique of the ASM-based semantic framework (Gargantini et al., 2009), which allows specifying the dynamic semantics of metamodel-based languages.

In the sequel, we first briefly introduce the semantic technique ([Section 6.1](#)), then the formal computational model of self-adaptive ASMs (Arcaini et al., 2017) where the MSL constructs will take semantics ([Section 6.2](#)), and finally the mapping from an MSL model to a self-adaptive ASM ([Section 6.3](#)). Such mapping, besides specifying the language semantics, provides a bridge toward the ASM-based back-end framework ASMETA¹⁰ (ASM mETAModeling) (Arcaini et al., 2011) for formal analysis of MSL models.¹¹

6.1. Semantic mapping technique

A metamodel-based language L has a well-defined semantics if a semantic domain S is identified and a semantic mapping $M_S: A \rightarrow S$ is provided (Harel and Rumpe, 2004) between the abstract syntax A (i.e., the metamodel of L) and S to give meaning to syntactic concepts of L in terms of the semantic domain elements. Although the semantic domain S and the mapping M_S can be described with varying degrees of formality, from natural language to rigorous mathematics, a clear and precise semantics requires both S and M_S to be defined in a formal way.

⁹ <http://www.arpalombardia.it>.

¹⁰ <http://asmeta.sourceforge.net/>.

¹¹ The ASMETA toolset is just an example of all possible target back-end frameworks. Further details are given in [Section 9](#).

Table 3

Mapping from MSL models into Self-adaptive ASMs.

MAPE element	ASM construct
Managed system Sys	Agent type $SysMda$ with program r_Sys modeling its behavior
Managing group Grp	Agent type $GrpMga$ with program r_Grp modeling its behavior
Component X ($X \in \{M, A, P, E\}$) of group Grp	Macro rule r_GrpX (called from program r_Grp) that models the behavior of component X . The rule contains a placeholder $<<TODO>>$ for indicating that the designer must supply an implementation.
Decentralized interaction $Grp1.X \rightarrow Grp2.Y[m_1, m_2]$	Functions modeling the interaction among agents of $Grp1Mga$ and $Grp2Mga$: - $fromGrp1toGrp2$ associating agents of $Grp1Mga$ to agents of $Grp2Mga$. The signature is $fromGrp1toGrp2: Grp1Mga \rightarrow Grp2Mga$ if $m_2=1$, and $fromGrp1toGrp2: Grp1Mga \rightarrow Powerset(Grp2Mga)$ if $m_2 \in \{\text{*}-ALL, \text{*}-SOME, \text{*}-ONE}\}. - fromGrp2toGrp1$ associating agents of $Grp2Mga$ to agents of $Grp1Mga$. The signature is $fromGrp2toGrp1: Grp2Mga \rightarrow Grp1Mga$ if $m_1=1$, and $fromGrp2toGrp1: Grp2Mga \rightarrow Powerset(Grp1Mga)$ if $m_1 \in \{\text{*}-ALL, \text{*}-SOME, \text{*}-ONE}\}. - sgnGrp1Grp2: Prod(Grp1Mga, Grp2Mga) \rightarrow Boolean$ modeling the <i>signals</i> exchanged among agents in order to trigger the interaction.
Centralized interaction $Grp.X \rightarrow Grp.Y[1, 1]$	Components rules X, Y of the same group Grp are called in a waterfall style by the agent of type $GrpMga$
Variation point semantics of multiplicity $m_1 \in \{\text{*}-ALL, \text{*}-SOME, \text{*}-ONE}\}$ in interaction $Grp1.X \rightarrow Grp2.Y[m_1, m_2]$	A derived function $startGrp2Y: Grp1Mga \rightarrow Boolean$ (read in rule r_Grp2Y) is used to combine the different values of the signals going from $Grp1Mga$ agents to a single $Grp2Mga$ agent $\$b$; function implementation depends on the variation point semantics: $\text{*}-ALL: (\text{forall } Sa \text{ in } fromGrp2toGrp1(\$b) \text{ with } sgnGrp1Grp2(Sa, \$b))$ $\text{*}-SOME: (\text{exist } Sa \text{ in } fromGrp2toGrp1(\$b) \text{ with } sgnGrp1Grp2(Sa, \$b))$ $\text{*}-ONE: (\text{exist unique } Sa \text{ in } fromGrp2toGrp1(\$b) \text{ with } sgnGrp1Grp2(Sa, \$b))$
Variation point semantics of multiplicity $m_2 \in \{\text{*}-ALL, \text{*}-SOME, \text{*}-ONE}\}$ in interaction $Grp1.X \rightarrow Grp2.Y[m_1, m_2]$	Rule r_Grp1X of agent $Grp1Mga$ sends signals to $Grp2Mga$ agents. The selected $Grp2Mga$ agents depend on the variation point semantics: $\text{*}-ALL: \text{to all associated agents}$ $\text{*}-SOME: \text{to a randomly selected subset of agents}$: $chooseone(\{Sa \in Powerset(Grp2Mga) not(isEmpty(Sa)) : Sa\})$ $\text{*}-ONE: \text{to a randomly selected agent}$: $chooseone(\{Sa \in Grp2Mga : Sa\})$
Concrete interactions in the configuration section	Declaration of agents, instances of agents types $SysMda$ and $GrpMga$, and initialization of interaction functions $fromGrp1toGrp2$ and $fromGrp2toGrp1$

Sometimes, in order to give the semantics of a language L , another helper language L' can be used, with L' endowed with a well-defined semantic domain S' and semantic mapping M'_S . L' can be exploited to define the semantics of L by (1) taking S' as semantic domain for L , i.e., $S' = S$; (2) introducing a *building function* $M_B: A \rightarrow A'$ which associates an element of A' to every construct of A ; (3) defining the *semantic mapping* $M_S: A \rightarrow S$ as $M_S = M'_S \circ M_B$. The function M_B can be built with different techniques (Gargantini et al., 2009).¹² We here exploit the *semantic mapping technique* where M_B is defined by using a model transformation language, and comes with a suitable tool that automatically applies M_B to any model m conforming to A in order to obtain a model m' conforming to A' .

Before applying the semantic mapping approach to define the semantics of the MSL language, we need to introduce the concept of *self-adaptive ASMs* that will play the role of the helper language.

6.2. Theoretical background on Self-Adaptive ASMs

ASMs (Börger and Raschke, 2018) are an extension of FSMs where unstructured control states are replaced by states comprising arbitrary complex data (i.e., domains of objects with functions defined on them), and transitions are expressed by transition rules describing how data (state function values saved into *locations*) change from one state to the next. ASM models can be easily read as “pseudocode over abstract data” which comes with a well defined semantics: At each run step, all transition rules are executed in parallel by leading to simultaneous (consistent) updates of a number of locations.

By exploiting the notion of *multi-agent ASM* – where each agent executes its own ASM in parallel with other agents’ ASMs and the agent’s *program* is the set of all transition rules of its own ASM –, in Arcaini et al. (2017) we provide the definition of *self-adaptive ASMs* as a multi-agent ASM where the set *Agents* is the disjoint union of the set *MgA* of *managing agents* and the set *MdA* of *managed agents*. Managing agents encapsulate the logic of self-adaptation, while managed agents encapsulate the system’s functional logic. Still in Arcaini et al. (2017), a MAPE loop (or interactive

MAPE loops) for an adaptation concern adj is defined as:

$$MAPE(adj) = \left\langle R_{adj}, \xrightarrow{adj}, K(adj) \right\rangle \quad (1)$$

R_{adj} is the set of transition rules, executed by managing agents, modeling the MAPE computations involved in the control loop; \xrightarrow{adj} is a relation on R_{adj} and is used to express MAPE computations interaction (e.g., M infers an A , which infers a P , which infers an E); $K(adj)$ is the knowledge (part of the locations of the self-adaptive ASM) used to keep the information necessary to enact and coordinate MAPE computations.

By the interaction relation definition, we can express a *decentralized* and *centralized* execution schema (also a mixed schema is possible) among MAPE computations. In the decentralized schema, rules in R_{adj} are executed by different agents, which interact with each other *indirectly* by sharing locations of the knowledge $K(adj)$ – and, therefore, rules are executed in different run steps. In the centralized schema, rules in R_{adj} are executed by the same managing agent either *indirectly*, or *directly* where each rule invokes the rule it is in interaction relation with – and, therefore, all rules are executed in one step (*waterfall style*).

6.3. Mapping MSL models into Self-Adaptive ASMs

Coming back to the semantic mapping technique introduced in Section 6.1, we here provide the definition of the building function M_B over MSL. As helper language, we assume the textual notation of the self-adaptive ASMs whose semantic domain and mapping are well defined (Gargantini et al., 2008; Arcaini et al., 2017).

The function M_B applied to an MSL model generates a self-adaptive ASM.¹³ Table 3 summarizes the mapping rules of the transformation process.

The agents set of the self-adaptive ASM is obtained by creating an agent in MdA for each managed system and an agent in MgA for each managing group of a concrete pattern.

The set of rules R_{adj} involved in the MAPE loop is built by creating a rule r_GrpX for each component X of each group Grp .

A decentralized interaction $Grp1.X \rightarrow Grp2.Y$ determines a relation between rules r_Grp1X and r_Grp2Y in \xrightarrow{adj} ; the correct interaction between the two rules is obtained by creating,

¹² Note that we have renamed here the building function M in Gargantini et al. (2009) with M_B to avoid misunderstanding with the monitor function M of a control loop.

¹³ Details on how we validated M_B are reported in Section 11.

in the knowledge $K(\text{adj})$, two functions, `fromGrp1toGrp2` and `fromGrp2toGrp1`, specifying the interacting agents, and a function `sgnGrp1Grp2` modeling the *signals* the agents use for establishing the communication. A centralized interaction between components X and Y of the same group Grp is obtained by directly invoking rule `r_GrpY` from `r_GrpX`.

The mapping also captures the desired semantics of * multiplicity; a source multiplicity of * kind is modeled by means of a formula (used as guard of a component rule) that requires that *all/some/exactly one* signal(s) must be received by the target agent in order to trigger the interaction. A target multiplicity of * kind, instead, is modeled by forcing the starting agent to write the signal to *all/some/exactly one* target agent(s). Note that we provide a default implementation of the choice of the target agents in case of *-SOME and *-ONE (i.e., they are selected randomly); however, the user should refine such choice with a more precise one, by considering the system requirements (see Section 7.1).

The generated self-adaptive ASM model is executable as coordination schema of MAPE components, but with empty implementation (marked by placeholders <>TODO>>) for component rules in R_{adj} (only standard writing and reading of signals is added to the model).

In order to model the behavior of the MAPE components performing a specific adaptation scenario, the generated ASM model has to be refined. This process is explained in Section 7.1.

Running example. As mapping example, Code 9 reports the ASM model (in the Asmetal notation) generated automatically from the MSL model shown in Code 6. According to the framework of self-adaptive ASMs presented in Section 6.2 (see Eq. 1) and the resulting ASM in Code 9, the MAPE loop for the adaptation concern HC is defined as $\text{MAPE}(\text{HC}) = \langle R_{\text{HC}}, \xrightarrow{\text{HC}}, K(\text{HC}) \rangle$, where

$$\begin{aligned} R_{\text{HC}} &= \{r_{\text{IntHCM}}, r_{\text{IntHCE}}, r_{\text{MainHCM}}, r_{\text{MainHCA}}, \\ &\quad r_{\text{MainHCP}}, r_{\text{MainHCE}}\} \\ \xrightarrow{\text{HC}} &= \{(r_{\text{IntHCM}}, r_{\text{MainHCM}}), \\ &(r_{\text{MainHCM}}, r_{\text{MainHCA}}), (r_{\text{MainHCA}}, r_{\text{MainHCP}}), \\ &(r_{\text{MainHCP}}, r_{\text{MainHCE}}), (r_{\text{MainHCE}}, r_{\text{IntHCE}})\} \\ K(\text{HC}) &= \{\text{fromIntHCMtoMainHCM}, \\ &\text{fromMainHCMtoIntHCM}, \text{sgnIntHCMMainHCM}, \\ &\text{fromMainHCEtoIntHCE}, \text{fromIntHCEtoMainHCE}, \\ &\text{sgnMainHCEIntHCE}\} \end{aligned}$$

7. Behavioral refinement and formal analysis of the smart home MAPE loops

MSL is a lightweight formalism for representing MAPE patterns and their instances. These models can be automatically mapped into self-adaptive ASMs, according to the semantic mapping shown in Section 6.3.

Once the MSL models are mapped into ASMs, the whole set of tools of the ASMETA framework can be used to provide early feedback about the correct loops execution as devised in Arcaini et al. (2017). However, in order to capture the intended behavior of the components realizing the adaptation concern, these formal models must be refined, and the starting point of this model refinement process is to replace the placeholders automatically generated by the model transformation, with effective ASM rules specifying the M,A,P,E computations. Effort for this refinement is domain-specific and has to be done manually. Moreover, the designer can also refine the standard implementation of * multiplicity by selecting the set of agents from/to which the signals must be received/sent. Moreover, the designer can also specify when the signals must be sent/received (all together or asynchronously); note that (as we have already discussed in Section 4.2), such temporal semantics can

not be specified in the MSL model and it can only be provided in the ASM model (thanks to the ASM operational semantics).

In this section, we first show a model refinement in order to completely specify a MAPE loop for an adaptation concern of the running case study, and then we show some possible uses of the ASMETA framework for formal analysis of the MAPE loops.

7.1. Model refinement

As an example of refinement, Code 10 reports an excerpt of the elements added in the refined model of the HC MAPE loop. A complete version of the ASM model for the HC concern is available online.

First, new signature elements (domains and functions) have been added to represent the knowledge for the OpenHAB items related to heating and windows of the managed home. For example, to accurately calculate how much heat is required to warm rooms, a monitored function `temperature` has been added to feed the model with the input temperature of the heated floors, i.e., the average value of all rooms temperature of a floor (as computed by the state aggregation function `AVG` for groups of numeric items in OpenHAB); indeed, in ASMs, monitored functions represent the inputs coming from the environment. The sensed temperature stored into the knowledge (by components M) is represented by a separate controlled function `temperature_Saved`. Similarly, for other sensed data.

Other knowledge functions have been added for the control process. For example, a function `desiredHeatingSetting` represents the adaptation decision made by component P, namely the heating speed mode for a floor (stopped, fairly hot, and very hot) chosen as final setting for the actuator in order to create enough heat and ensure the floor is heated as set on the floor's thermostat (the monitored function `temperature_Setpoint`). The final setting for the actuator is realized by the component E by assigning the planned value to the function `heating_Setting` (read by the managed agent). This last is the value to send through an OpenHAB command (the actuator or effector) to the item `Heating_Setting_GF` or `Heating_Setting_FF`, depending on the floor, of the managed home. Similarly, a group of functions have been added for the hold-closed functionality to keep windows closed when heating is on.

Moreover, a flag (the boolean-valued function `loopHCCCompleted`) has been added for the coordination of different execution instances of the compound MAPE loop; here we assume that different executions of the same compound loop are done one at a time to avoid to handle another adaptation cycle while one is still executing and the loop knowledge is in a transient state.

Then, some derived boolean-valued functions generated and used as guards for triggering the MAPE rules (components) have been refined. For example, the function `startIntHCM` for triggering the component M of an interface group has been defined to be exactly the logical boolean-valued function `loopHCCCompleted` in order to start a new execution of the loop only when the previous one (if any) completed.

Finally, placeholders of component rules and cleaning rules have been refined. For example, in `r_IntHCE`, the final setting for heating and windows, respectively, is done by updating the functions `heating_Setting` and `windows_Setting` according to the planned decisions.

7.2. Simulation-based analysis

The first kind of analysis that a designer can perform is model validation by simulation. Through simulation, the developer can interact with the model and get an early understanding of its be-

<pre> asm MySmartHomeHCPplus import StandardLibrary signature: //HC.MAPE domain HeatingMdA subsetof Agent domain AirMdA subsetof Agent domain MainHCMgA subsetof Agent domain IntHCMgA subsetof Agent derived startMainHCM: MainHCMgA -> Boolean derived startMainHCA: MainHCMgA -> Boolean derived startMainHCP: MainHCMgA -> Boolean derived startMainHCE: MainHCMgA -> Boolean controlled heatingManagedByIntHC: IntHCMgA -> HeatingMdA controlled airManagedByIntHC: IntHCMgA -> AirMdA derived startInHCM: IntHCMgA -> Boolean derived startInHCE: IntHCMgA -> Boolean //I: IntH.C.M -> MainH.C.M /*- SOME,1] controlled sgnInHCMMainHCM: Prod(IntHCMgA, MainHCMgA) -> Boolean controlled fromInHCMtoMainHCM: IntHCMgA -> MainHCMgA controlled fromMainHCMtoInHCM: MainHCMgA -> Powerset(IntHCMgA) //I: MainH.C.E -> IntH.C.E [/*- SOME] controlled sgnInHCEInHCE: Prod(IntHCMgA, IntHCMgA) -> Boolean controlled fromMainHCEtoInHCE: MainHCMgA -> Powerset(IntHCMgA) derived orSelectorfromMainHCEtoInHCE: MainHCMgA -> Powerset(IntHCMgA) controlled fromInHCEtoMainHCE: IntHCMgA -> MainHCMgA //MySmartHomeHC static hs_ff: HeatingMdA static hs_gf: HeatingMdA static ws_gf: AirMdA static ws_ff: AirMdA static main_hc: MainHCMgA static int_hc_gf: IntHCMgA static int_hc_ff: IntHCMgA definitions: function startMainHCM(\$b in MainHCMgA) = (exist \$a in fromMainHCMtoInHCM(\$b) with sgnInHCMMainHCM(\$a, \$b)) function startMainHCA(\$b in MainHCMgA) = true function startMainHCP(\$b in MainHCMgA) = true function startMainHCE(\$b in MainHCMgA) = true function startInHCM(\$b in IntHCMgA) = true function startInHCE(\$b in IntHCMgA) = sgnInHCEInHCE(fromInHCEtoMainHCE(\$b), \$b) function orSelectorfromMainHCEtoInHCE(\$b in MainHCMgA) = chooseone({\$a in Powerset(IntHCMgA) not(isEmpty(\$a))}: \$a) rule r.Heating = skip //<< TODO >> rule r.Air = skip //<< TODO >> rule r.CleanUp_InHCM = skip //<< TODO >> rule r.IntHCM = if startInHCM(self) then par skip //<< TODO >> if not sgnInHCMMainHCM(self,fromInHCMtoMainHCM(self)) then sgnInHCMMainHCM(self,fromInHCMtoMainHCM(self)) := true endif r.CleanUp_InHCM[] endpar endif rule r.CleanUp_InHCE = sgnInHCEInHCE(fromInHCEtoMainHCE(self), self) := false rule r.IntHCE = if startInHCE(self) then par skip //<< TODO >> r.CleanUp_InHCE[] endpar endif rule r.IntHC = par r.IntHCM[] r.IntHCE[] endpar rule r.CleanUp_MainHCM = forall \$a in fromMainHCMtoInHCM(self) do sgnInHCMMainHCM(\$a, self) := false </pre>	<pre> rule r.MainHCM = if startMainHCM(self) then par skip //<< TODO >> r.MainHCA[] r.CleanUp_MainHCM[] endpar endif rule r.CleanUp_MainHCA = skip //<< TODO >> rule r.MainHCA = if startMainHCA(self) then par skip //<< TODO >> r.MainHCP[] r.CleanUp_MainHCA[] endpar endif rule r.CleanUp_MainHCP = skip //<< TODO >> rule r.MainHCP = if startMainHCP(self) then par skip //<< TODO >> r.MainHCE[] r.CleanUp_MainHCP[] endpar endif rule r.CleanUp_MainHCE = skip //<< TODO >> rule r.MainHCE = if startMainHCE(self) then par skip //<< TODO >> forall \$a in orSelectorfromMainHCEtoInHCE(self) do sgnInHCEInHCE(self, \$a) := true r.CleanUp_MainHCE[] endpar endif rule r.MainHC = r.MainHCM[] main rule r.mainRule = forall \$a in Agent with true do program(\$a) default int \$o: function sgnInHCMMainHCM(\$a in IntHCMgA, \$b in MainHCMgA) = false function sgnInHCEInHCE(\$a in MainHCMgA, \$b in IntHCMgA) = false function fromInHCMtoMainHCM(\$a in IntHCMgA) = switch(\$a) case int_hc_gf: main_hc case int_hc_ff: main_hc endswitch function fromMainHCMtoInHCM(\$a in MainHCMgA) = switch(\$a) case main_hc: {int_hc_gf, int_hc_ff} endswitch function fromMainHCEtoInHCE(\$a in MainHCMgA) = switch(\$a) case main_hc: {int_hc_gf, int_hc_ff} endswitch function fromInHCEtoMainHCE(\$a in IntHCMgA) = switch(\$a) case int_hc_gf: main_hc case int_hc_ff: main_hc endswitch function airManagedByIntHC(\$x in IntHCMgA) = switch(\$x) case int_hc_gf: ws_gf case int_hc_ff: ws_ff endswitch function heatingManagedByIntHC(\$x in IntHCMgA) = switch(\$x) case int_hc_gf: hs_gf case int_hc_ff: hs_ff endswitch agent IntHCMgA: r.IntHC[] agent HeatingMdA: r.Heating[] agent MainHCMgA: r.MainHCM[] agent AirMdA: r.Air[] </pre>
--	--

Code 9. ASM model of the Heating Comfort (HC) MAPE loop.

havior; already at this stage, trivial errors can be discovered. The ASMETA framework provides three ways of simulating the model:

- (1) interactive textual simulation by the AsmetaS tool ([Gargantini et al., 2008](#)) in which the user interactively sets the values of monitored functions and inspects the simulation output for checking correctness;
- (2) graphical animation by the AsmetaA tool ([Bonfanti et al., 2018](#)) that provides a visualization of the output of the AsmetaS simulator. The tool facilitates the understanding of the simulation, as it provides a more compact visualization of the states and shows which locations have changed be-

tween two consecutive states and which have been remain unchanged;

- (3) scenario-based simulation by the AsmetaV ([Carioni et al., 2008](#)) tool in which the user specifies in a *scenario* (similar to a test case) the interaction with the model: (s)he can set values for monitored functions, enforce a simulation step, and check that some predicates hold in some given states.

As example of model validation, we here use the animator AsmetaA. [Fig. 6](#) shows a simulation trace in which the user sets, in the initial state, the status of the house: at each floor the heater is turned on (`heatingStatus(hs_gf)`) and

```

asm MySmartHomeHcplus_refined
import StandardLibrary
signature:
...
//domains added in refinement
enum domain HeatingSetting = {FAIRLY_HOT | VERY_HOT | STOPPED}
enum domain HeatingStatus = {ON|OFF}
enum domain WindowStatus = {OPEN|CLOSED}
domain Temperature subsetof Integer
...
//functions added in refinement
//knowledge for the managed smart home items (heating and windows)
monitored temperature: HeatingMdA -> Temperature
monitored temperature_Setpoint: HeatingMdA -> Temperature
monitored temperature_Midpoint: HeatingMdA -> Temperature
monitored windowsStatus: AirMdA -> WindowStatus
monitored heatingStatus: HeatingMdA -> HeatingStatus
controlled temperatureSaved: IntHCMgA -> Temperature
controlled temperature_SetpointSaved: IntHCMgA -> Temperature
controlled temperature_MidpointSaved: IntHCMgA -> Temperature
controlled windowsStatusSaved: IntHCMgA -> WindowStatus
controlled heatingStatusSaved: IntHCMgA -> HeatingStatus
derived heatingsON: Boolean
controlled desiredHeatingSetting: IntHCMgA -> HeatingSetting
controlled desiredWindowsStatus: IntHCMgA -> WindowStatus
derived computeHeatingSetting : IntHCMgA -> HeatingSetting
controlled heating_Setting: HeatingMdA -> HeatingSetting
controlled windows_Setting: AirMdA -> WindowStatus

//Function for the coordination of the overall compound MAPE loop; they are
//used to strictly sequence different executions of the same loop
controlled loopHCCompleted: IntHCMgA -> Boolean

definitions:
//Refined for sequencing different MAPE loops executions
function startInthCM($b in IntHCMgA) =
loopHCCompleted($b)

//Refined to check if adaptation is required
function startMainHCA($b in MainHCMgA) =
(heatingsON and (exist $a in fromMainHCMtoInthCM($b)
    with neq(desiredHeatingSetting($a),computeHeatingSetting($a))) or
(heatingsON and (exist $c in fromMainHCMtoInthCM($b)
    with eq(windowsStatusSaved($c),OPEN)))
...
//Added in refinement to compute the desired heating setting
function computeHeatingSetting($a in IntHCMgA) =
let ($t = temperatureSaved($a)) in
    if $t < temperature.MidpointSaved($a) then //e.g. < 10
        VERY_HOT
    else
        if $t < temperature_SetpointSaved($a) then //e.g. < 18
            FAIRLY_HOT
        else
            STOPPED
        endif
    endif
endlet

//Added in refinement
function heatingsON = (exist $a in IntHCMgA with heatingStatusSaved($a) = ON)

//Added in refinement. It saves the temperature—related data of a room/zone (sensed by
//the managed thermostat) and windows status into the knowledge
rule r_SaveSensorsData =
par
    temperatureSaved(self) := temperature(heatingManagedByInthC(self))
    temperature_SetpointSaved(self) := temperature_Setpoint(heatingManagedByInthC(self))
    temperature_MidpointSaved(self) := temperature_Midpoint(heatingManagedByInthC(self))
    windowsStatusSaved(self) := windowsStatus(airManagedByInthC(self))
    heatingStatusSaved(self) := heatingStatus(heatingManagedByInthC(self))
endpar

//** Interface group MAPE components' behavior **
rule r_CleanUp_InthCM =
loopHCCompleted(self) := false //added in refinement

rule r_InthCM =
if startInthCM(self) then
par
    r_SaveSensorsData[] //added in refinement
    if not sgnlnthCMMainHCM(self,fromInthCMtoMainHCM(self)) then
        sgnlnthCMMainHCM(self,fromInthCMtoMainHCM(self)) := true endif
    r_CleanUp_InthCM[]
endpar
endif

rule r_CleanUp_InthCE =
par
    sgnMainHCEInthCE(fromInthCEtoMainHCE(self), self) := false
    loopHCCompleted(self) := true //added in refinement
endpar

rule r_InthCE =
if startInthCE(self) then
par //Added in refinement: final setting for the actuators
    heating_Setting(heatingManagedByInthC(self)) := desiredHeatingSetting(self)
    windows_Setting(airManagedByInthC(self)) := desiredWindowsStatus(self)
    r_CleanUp_InthCE[]
endpar
endif

//** Main group MAPE components' behavior **
rule r_MainHCM =
if startMainHCM(self) then
par
    r_MainHCA[]
    r_CleanUp_MainHCM[]
endpar
endif

rule r_CleanUp_MainHCA =
//if no adaptation was necessary, cleaning of loop completion flags is done here
forall $a in fromMainHCMtoInthCM(self) with loopHCCompleted($a) = false do
    loopHCCompleted($a) := true

//Refined
rule r_MainHCA =
if startMainHCA(self) then
    r_MainHCP[]
else
    r_CleanUp_MainHCA[]
endif

//Refined
rule r_MainHCP =
if startMainHCP(self) then
par
    forall $a in fromMainHCMtoInthCM(self) do
        par
            desiredHeatingSetting($a) := computeHeatingSetting($a)
            if (heatingsON and windowsStatusSaved($a) = OPEN) then
                desiredWindowsStatus($a) := CLOSED endif
        endpar
        r_MainHCE[]
    endpar
endif

rule r_MainHCE =
if startMainHCE(self) then
forall $a in fromMainHCMtoInthCM(self) do
    par
        sgnMainHCEInthCE(self, $a) := true
    endif
...
default init s0:
...
//Added in refinement
function desiredHeatingSetting($a in IntHCMgA) = STOPPED
function loopHCCompleted ($a in IntHCMgA) = true

```

Code 10. Excerpt of the refined ASM model of the Heating Comfort (HC) MAPE loop.

heatingStatus(hs_ff)), at least one window is open on ground floor (windowsStatus(hs_gf)), all windows are closed on first floor (windowsStatus(hs_ff)), and the temperature on both floors is 18 (temperature(hs_gf) and temperature(hs_ff)) with a desired temperature of 20 (temperature_Setpoint(hs_gf) and temperature_Setpoint(hs_ff)). Starting from this setting, the MAPE loop for HC concern decides to perform adaptation and, in three steps, it closes the window at the ground floor (windowsSetting(hs_gf)) and sets the heating system to FAIRLY_HOT on both floors (heatingSetting(hs_gf) and heatingSetting(hs_ff)). In the online repository, we also report an AsmetaV scenario automatizing this simulation trace.

7.3. Model checking-based analysis

Although simulation is very useful and it often allows to quickly find trivial model faults, more subtle faults may be more difficult

to discover and, therefore, more advanced analysis techniques exploring the whole state space need to be applied.

As first technique, the designer can specify application-dependant temporal properties (either CTL or LTL) over the model and check them using the AsmetaSMV model checker (Arcaini et al., 2010a). For example, for the HC concern of the running case study, we specified the following properties regarding the adaption correctness:

$$\begin{aligned}
& G((heatingsON \text{ and } temperatureSaved(int_hc_gf) < \\
& \quad \quad \quad \text{temperature_MidpointSaved(int_hc_gf)}) \\
& \quad \quad \quad \text{implies } F(\text{heating_Setting}(hs_gf) = \text{VERY_HOT})) \\
& G((heatingsON \text{ and } temperatureSaved(int_hc_ff) < \\
& \quad \quad \quad \text{temperature_MidpointSaved(int_hc_ff)}) \\
& \quad \quad \quad \text{implies } F(\text{heating_Setting}(hs_ff) = \text{VERY_HOT})) \\
& G((heatingsON \text{ and } windowsStatusSaved(int_hc_gf) = \text{OPEN}) \\
& \quad \quad \quad \text{implies } F(\text{windows_Setting}(hs_gf) = \text{CLOSED})) \\
& G((heatingsON \text{ and } windowsStatusSaved(int_hc_ff) = \text{OPEN}) \\
& \quad \quad \quad \text{implies } F(\text{windows_Setting}(hs_ff) = \text{CLOSED}))
\end{aligned}$$

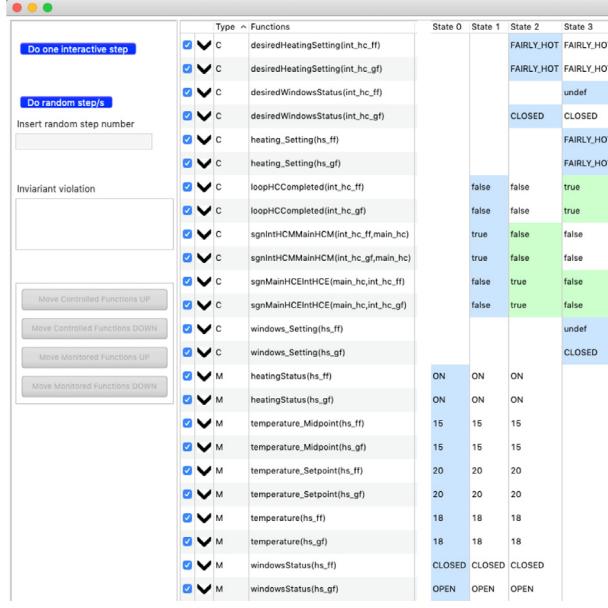


Fig. 6. Simulation through the AsmetaA tool.

The first two properties check that, whenever the saved temperature value is below the desired value in a given floor, the heating system will be eventually set to the maximum power (i.e., VERY_HOT). The last two properties, instead, check that if, in a given floor, at least one window is open and the heating system is turned on, the windows will be eventually closed. Similar application-dependant properties have been specified for the other two adaptation concerns and can be found in the online repository.

The tool AsmetaMA (Arcaini et al., 2010b) of the ASMETA framework, instead, allows to check application-independent properties that any model should guarantee (called *meta-properties*). The tool checks 7 meta-properties related to *minimality*, *completeness*, and *consistency*. For example, the consistency meta-property MP1 checks that there are no *inconsistent updates* (Börger and Raschke, 2018), i.e., no location is updated to two different values at the same time. We applied the model reviewer to all the models developed for the running case study and it proved to be useful on the model combining the AQ and HC concerns; indeed, for this model, AsmetaMA found that MP1 is violated as location `windows_Setting(aqs_gf)` (similarly, also location `windows_Setting(aqs_ff)`) can be updated simultaneously to OPEN and CLOSED when the MAPE loop for AQ tries to open the window to refresh the air, and the MAPE loop for HC tries to close it to avoid that the windows are open when the heating system is on. Thanks to this analysis result, we realized that the original MSL design was not accurate enough and it was not implementing a proper coordination schema between the MAPE loops handling the conflicting concerns; this lead us to redesign the MSL model as explained in the next section.

8. Analysis-driven re-design of the smart home case study

To resolve the conflict arising between the goals of the MAPE loops for AQ and HC concerns in managing the windows opening/closure, the MAPE loops need to cooperate and coordinate actions to avoid interferences and provide certain guarantees about adaptations. To this purpose, the two MAPE loops have been composed to realize a control hierarchy according to the abstract pattern definition already presented in [Code 4, Section 4](#). [Code 11](#) shows the corresponding concrete pattern and its instance for the compound concern AQ_HC. A higher MAPE group `high_aq_hc` operates at

```

import HierarchicalControlMAPE
concrete pattern Hierarchical_AQ_HC_MAPE
    concretizationOf HierarchicalControlMAPE {
        //Heating sub-system
        system SysH: HierarchicalControlMAPE.SysOne
        //Air (windows) sub-system
        system SysA: HierarchicalControlMAPE.SysTwo
    }

    group MainHC: HierarchicalControlMAPE.Intermediate
    group IntHC: HierarchicalControlMAPE.InterfaceOne
    group MainAQ: HierarchicalControlMAPE.InterfaceTwo
    group HighAQ_HC: HierarchicalControlMAPE.High
}

configuration MySmartHomeAQ_HC instanceOf
    Hierarchical_AQ_HC_MAPE {
        hs_ff: Hierarchical_AQ_HC_MAPE.SysH //heating Fst Floor
        hs_gf: Hierarchical_AQ_HC_MAPE.SysH //heating Gr Floor
        aqs_gf: Hierarchical_AQ_HC_MAPE.SysA //air Gr Floor
        aqs_ff: Hierarchical_AQ_HC_MAPE.SysA //air Fst Floor

        //group for the concern Air Quality
        main_aq: Hierarchical_AQ_HC_MAPE.MainAQ {
            managedSys aqs_gf, aqs_ff
            components m_aq:M, a_aq:A, p_aq:P, e_aq:E
        }

        //main group for the concern Heating Comfort
        main_hc: Hierarchical_AQ_HC_MAPE.MainHC {
            components m_hc:M, a_hc:A, p_hc:P, e_hc:E
        }

        //interface group for the ground floor heating and windows
        int_hc_gf: Hierarchical_AQ_HC_MAPE.IntHC {
            managedSys hs_gf, aqs_gf
            components m_hc:M, e_hc:E
        }

        //interface group for the first floor heating and windows
        int_hc_ff: Hierarchical_AQ_HC_MAPE.IntHC {
            managedSys hs_ff, aqs_ff
            components m_hc:M, e_hc:E
        }

        //higher group
        high_aq_hc: Hierarchical_AQ_HC_MAPE.HighAQ_HC {
            managedGrp main_hc, main_aq
            components m_hc:M, a_hc:A, p_hc:P, e_hc:E
        }

        //interactions
        main_aq.m_aq -> main_aq.a_aq
        main_aq.a_aq -> main_aq.p_aq
        main_aq.p_aq -> main_aq.e_aq
        main_hc.m_hc -> main_hc.a_hc
        main_hc.a_hc -> main_hc.p_hc
        main_hc.p_hc -> main_hc.e_hc
        int_hc_gf.m_hc -> main_hc.m_hc
        int_hc_ff.m_hc -> main_hc.m_hc
        main_hc.e_hc -> int_hc_gf.e_hc
        main_hc.e_hc -> int_hc_ff.e_hc
        high_aq_hc.a_h -> high_aq_hc.e_h
    }
}

```

Code 11. MAPE loop for the concern AQ_HC.

a longer time scale with a more global/strategic vision and may prioritize actions of the loops `main_aq` for AQ and `main_hc` for HC at lower layers that operate at a short time scale, guaranteeing timely adaptation concerning the part of the system under their direct control. The possibility of interference between the two loops is now made explicit at MSL model level (see [Code 11](#)) by specifying the air (windows) subsystems (`aqs_gf` and `aqs_ff`) of the two floors as managed subsystems of both the two loops for AQ and HC (`aqs_gf` is managed by groups `int_hc_gf` and `main_aq`, and `aqs_ff` is managed by groups `int_hc_ff` and `main_aq`).

Starting from the self-adaptive ASM generated from this new MSL model for AQ_HC, we reused part of the behavior specifica-

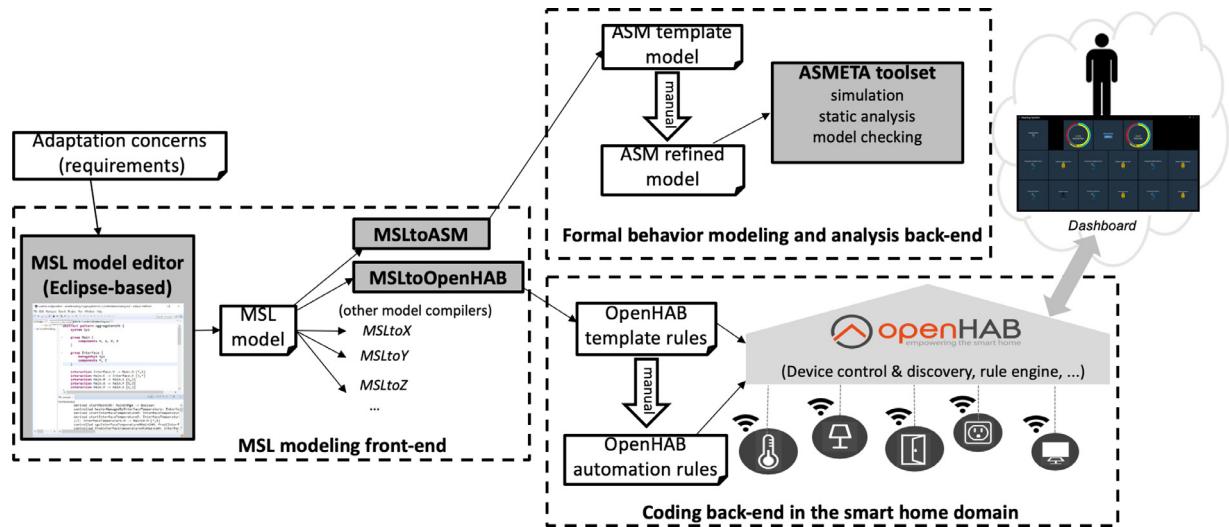


Fig. 7. Overview of the MSL-centric framework.

```

rule r_HighAQ_HCE =
  if startHighAQ_HCE(self) then
    par
      if (isDef(desiredWindowsStatus(main_aq))) then
        desiredWindowsStatus(self):= desiredWindowsStatus(main_aq)
      else desiredWindowsStatus(self):= undef endif
      if (not(sgnHightoMainHC_E(main_hc))) then sgnHightoMainHC_E(main_hc):= true endif
      if (not(sgnHightoMainAQ_E(main_aq))) then sgnHightoMainAQ_E(main_aq):= true endif
      r_CleanUp_HighAQ_HCE]
    endpar
  endif

```

Code 12. Excerpt of the ASM model of the AQ_HC MAPE loop.

tion for the MAPE components of the two loops AQ and HC and we further refined and extended it to capture the intended cooperation and coordination as realized by the new arbiter agent for the higher group. The complete ASM model for the compound concern AQ_HC is available online. [Code 12](#) reports an excerpt of the component E (rule `r_HighAQ_HCE`) of the higher group showing how decision is made to resolve the conflict. It is just a way to solve it; essentially, if the location value `desiredWindowsStatus` has been established for AQ, discordant or not in value with that for HC, the last decision is made by the higher agent in favour of the desired setting of AQ, and communicated to the components E of the managed lower groups. The lower agents' components E (not shown in [Code 12](#)) will then consider the decision of the higher agent in actuation.

9. Tools implementation around MSL

Around the MSL language, as long-term plan, we are developing a pattern-oriented framework for the design of the managing layer in self-adaptive systems. The idea is to have a framework centered around MSL, and extensible by including external components for different activities around the models. The overall framework is depicted in [Fig. 7](#).

The MSL editor, together with its parser and validator (presented in [Section 4.3](#)), makes up the modeling front-end. The Xtext/EMF approach MSL has been built on, facilitates the development of compilers (i.e., *model generators*) toward other (back-end) frameworks by using principles and tools of the model-driven engineering.

The model generator MSL2ASM, for example, automatically translates MSL models into self-adaptive ASMs. It has permitted the integration of the ASMETA toolset into the MSL-centric framework as back-end for formal validation and verification of the MSL

models (some of the ASMETA features have been presented in [Section 7](#)). MSL2ASM has been developed in Java using a Model-to-Text (M2T) approach. According to the mapping rules synthesized in [Table 3](#), the generator visits the MSL.ecore metamodel instance – i.e., the abstract syntax tree (AST) of an MSL model¹⁴ – to produce the corresponding ASM model. The generator can be installed as eclipse plugin together with the MSL editor.

Recently, another component has been integrated into the framework as implementation back-end of MSL models in the context of the smart home systems. A model generator MSL2OpenHAB has been implemented by using the same M2T approach and automatically translates MSL models into a set of OpenHab items and a set of *Event-Condition-Actions* rules of the OpenHAB automation rule engine. A preliminary version of this component, whose presentation is out of the scope of this paper, has been presented in [Arcaini et al. \(2019\)](#). This component increases the use of OpenHAB both as platform to implement the managed layer of a self-adaptive system (as used in this paper) and to implement the managing layer.

10. Assessment of the MSL modeling language

In this section, we discuss about the assessment of the MSL language concerning its support for the requirements presented in [Section 4.1](#). We discuss them individually, specifying also a degree of support (low, medium, and high).

Model purpose. (High support). To achieve the model purpose expressed for MSL and the vision of an MSL-centric modeling framework, we adopted a loose-form of modeling/coding language composition via sharing a set of core concepts/abstractions with the same semantics. Tools that manipulate models/code do not exchange information explicitly, but reason about artifacts related to shared semantic concepts/abstractions ([Clark et al., 2014](#)). The MSL framework realizes this loose-form of language composition based on a core DSL of the *model space* (i.e., MSL) and on *implicit modeling* ([Kautz et al., 2018](#)) in the DSLs of the target *solution spaces* (such as the ASM-based ASMETA formal analysis toolset). Essentially, we employ ground MSL models for the structuring of the adaptation loops activities as composable units w.r.t. the adaptation concerns; we implicitly model the loops structure via model/code generators towards specific solution spaces; and via incremental refinement in

¹⁴ This AST is the EMF.ecore model of MSL used as the in-memory objects representation of any parsed MSL text file.

the target solution space, we complete the behavioral specification of the loops activities in an operational form.

The ASM-based ASMETA specification and analysis toolset was the first back-end module introduced in the MSL framework to enable formal specification of MAPE-K loops behavior (the internal MAPE components behavior) and its formal analysis (validation and verification).

A second back-end module was recently introduced (Arcaini et al., 2019) for the generation of implementation code in the rule-based programming environment of the smart home platform OpenHAB. Due to the fact that the two back-end modules are quite diverse (syntactically and semantically), we think that the degree of support is high. However, to completely assess the level of fulfilment of such a requirement, more translation toward other back-end frameworks are needed.

Separation of concerns. (High support). According to such a best practice, in MSL we support separation of adaptation concerns and the design of their MAPE-K loop models separately as instances of specific MAPE patterns.

Principle of Least Knowledge. (High support). A single or structured MAPE-K loop inherently satisfies such a principle, since MAPE-K components interact according to a pre-fixed sequence of interactions and communicate data via a shared knowledge. So MAPE components do not depend on the internal structure of other MAPE components, and therefore they can be replaced without impact on the others.

Minimize upfront design (Medium support). Thanks to the support of the separation of adaptation concerns and to the definition of a separate (possibly structured) MAPE-K loop for each of them, MSL provides a way to create just enough upfront design that does not become a burden for designers. Though we do not mention here any development methodology for self-adaptive systems explicitly, we believe that, since MSL allows the creation of a minimal initial design of the MAPE-K loops structure, MSL thus provides only required details as needed to support the analysis and design in agile iterations.

Pattern-oriented design. (High support). MSL fully supports the pattern-oriented approach proposed in Weyns et al. (2013) to the design of the adaptation layer in terms of recurring structures of MAPE-K loops. Design experts can use MSL built-in patterns but also add, extend, and modify patterns to tailor them to their own needs. MSL allows both the description and concrete use of the decentralized MAPE patterns given in Weyns et al. (2013), and also the definition of new ones just as they are borne from experience or as result from a pattern composition.

Model composability. (Medium support). Currently, we support MSL model composition at the level of the definition of the abstract patterns, essentially by integrating the definition of their corresponding abstract patterns into one abstract pattern. An example of such composition was presented in Section 8 where we introduced a new pattern to compose two existing MAPE loops in a control hierarchy. As future work, we intend to extend the MSL language with patterns composition operators applicable during the concretization phase of a pattern instantiation.

Concerning the problem of assuring the dependability and trustworthiness after the composition, thanks to the model projection into the ASM-based ASMETA space, we are able to provide assurance on the functional correctness of the MAPE-K loop behavior as resulting from the composition. Assurance of quality properties of the adaptation layer itself is not yet supported by the MSL framework, but recently we have been investigating on how to support it by introducing the notion of *self-accounting* as self-* property and presenting an inductive method that, based on the structure of the MAPE patterns, evaluates the cost of the adaptation logic (in terms, for example, of latency time and availability) (Mirandola et al., 2019).

11. Threats to validity

Following the approach outlined in Runeson and Höst (2009); Wohlin et al. (2012) we discuss potential threats to the validity of the proposed language and approach.

To smooth the *construct validity* threat, we have designed the whole language following proven principle in software engineering as, the separation of concerns, the pattern oriented design, and the model composability (see Section 4.1). The fulfillment of these principles is thoroughly discussed in Section 10.

A threat to *internal validity* is that the ASM generator MSL2ASM could be not correct and, therefore, the produced ASMs would not be self-adaptive ASMs that correctly describe the MAPE patterns of the corresponding MSL models. To deal with this threat, we check that the produced ASM models are indeed correct by applying the “MAPE-K Correctness Verification” approach described in Arcaini et al. (2017). The approach generates a set of LTL properties starting from an ASM model *A* that allegedly implements a self-adaptive ASM; each property is the LTL characterization of an indirect interaction $X \rightarrow Y$ of the MAPE loop that should be implemented: intuitively, it states that if *Y* occurs, *X* occurred in the past and triggered *Y*. If all the generated LTL properties are verified, the ASM *A* is indeed a self-adaptive ASM. We applied this technique to all the generated ASMs¹⁵ and it confirmed that they are self-adaptive ASMs. Although this is not a proof of total correctness, it provides a good degree of confidence that the generator MSL2ASM is correct.

A threat to *external validity* is the representativeness of the case study. In fact, as discussed above, MSL has been assessed through the development of a smart home case study, therefore only a limited set of features has been investigated. To smooth this threat, we have applied MSL to another case study from the automotive domain. In particular, we modeled the adaptive features (the adaptive high beam headlights and the adaptive cruise control) of the *Adaptive Exterior Light and Speed Control Systems*.¹⁶ Further details are publicly available.¹⁷ Another possible threat to external validity is the usability of the language. To smooth this threat, the syntax of the proposed language is intuitive and self-explanatory. We have also performed an initial qualitative evaluation with a small set of colleagues obtaining a positive feedback. As a future work, we plan to include external experts (students and professionals) in a controlled experiment which will design a fully-featured application in the context of an ongoing research project. Hence, a quantitative evaluation of the experience in adopting MSL can be performed to derive some meaningful statistics about the language usability.

Regarding the *reliability validity* threat, researchers willing to adopt MSL to model their managing systems in accordance to the patterns in Weyns et al. (2013) are supported by the fact that the existing MAPE patterns are now pre-defined and publicly available in MSL. Besides, the language itself has been conceived at a fine granularity level that allows an easy definition of new patterns that can extend the pattern library. Moreover, to provide graphical views of MSL models (similarly to those in Weyns et al. (2013)), we are in the process of implementing also a graphical visualizer.

12. Related work

We give here a brief overview of selected works that are related to notations and tools for explicitly modeling MAPE loops (and patterns) that we identified as the most relevant to the context of this work.

¹⁵ Models used for validation of MSL2ASM are reported in <https://github.com/fmselab/msl/tree/master/org.xtext.mslasmgenerator/modelsForValidation>.

¹⁶ <https://abz2020.uni-ulm.de/case-study>.

¹⁷ <https://github.com/fmselab/msl/tree/master/examples/CarSystem>.

Languages. Contributions in Iglesia and Weyns (2015); Calinescu et al. (2018) exploit the use of a network of Timed Automata to specify the behavior of MAPE components, and the Uppaal model checker for property verification. A development methodology, called ENTRUST, supports the systematic engineering and assurance of self-adaptive systems. In ENTRUST, a template-based approach to the design and verification of a “specific family” of self-adaptive systems is used, namely a target domain of distributed applications in which self-adaptation is used for managing resources for robustness and openness requirements via adding/removing resources from the system. In Hebig et al. (2010), a UML profile is proposed to model control loops as first-class entities when architecting software with UML. The UML profile supports modeling of interactions between coarse-grained controllers, while the MSL language aims at modeling finer-grained interactions between the MAPE components. A dedicated model language, called Stitch, has been presented in Cheng and Garlan (2012), to support adaptation strategies in self-adapting software architectures. The language, used in the Rainbow framework (Garlan et al., 2004), defines adaptation strategies as decision trees and allows the explicit representation of quality of services objectives, the selection of strategies that optimize the system utility considering also the presence of potential timing delays and outcome uncertainty.

With respect to these languages, that do not explicitly model MAPE loops, in MSL, we elevate them to first-class entities for structuring the adaptation logic of any self-adaptive system in the early design phases and for fostering (in a broad sense) *pattern-oriented modeling*.

Tools. SOTA (State Of The Affairs) (Abeywickrama et al., 2013), supported by the Eclipse plug-in SimSOTA, is a goal-oriented approach for modeling, simulating and validating self-adaptive systems. Unlike our approach, SOTA adopts a semi-formal notation, namely UML activity diagrams, to model the behavior of feedback loops. The framework ACTRESS (Kříkava et al., 2014) is grounded on an actor-oriented component meta-model and provides support for structural modeling of feedback loops, model well-formedness checking (through structural OCL or Xbase invariants), and generation of Java-like code for the actor- and JVM-based runtime platform Akka. Both SOTA and ACTRESS do not support pattern modeling and do not adopt a formal notation, as self-adaptive ASMs, for the behavior specification and verification of MAPE components.

ActivFORMS (Active FORmal Models for Self-adaptation) (Iftikhar and Weyns, 2017) is a virtual machine to realize self-adaptation with guarantee. Timed automata are used to design the MAPE-K loops accomplishing given adaptation goals. These models are verified off-line and at run-time, and can be updated on the fly to support changing goals. ActivFORMS shares with us a formal description of the adaptation for verification, but it does not support coding of the models. CYPHEF (CYber-PHysical dEvelopment Framework) (D'Angelo et al., 2017) provides a graphical notation for modeling the control architecture of a cyber-physical system by MAPE loop patterns. Differently from our approach, CYPHEF does not provide support for the specification and formal verification of the MAPE components' behavior.

Despite similarities and differences with our approach, all the works mentioned above can be used as back-end frameworks to complement and complete for different purposes the adaptation logic design started in MSL.

13. Conclusion and future work

We proposed the textual language MSL for defining and instantiating MAPE patterns in structuring the adaptation logic of self-adaptive systems. The language provides a textual counterpart of the graphical notation originally presented in Weyns et al.

(2013) for specifying MAPE patterns, but never developed and exposing a number of ambiguities. A semantic mapping and a model generator from MSL to self-adaptive ASMs has been also presented. It provides a connection to the modeling back-end of ASMs for specifying and analysing the behavior of instances of MAPE patterns formally. The language has been used to model the adaptation layer of case studies in the area of smart home and vehicle automation.

MSL can be used to model complex composite MAPE loops structures. We support a repository where all patterns devised in Weyns et al. (2013) are available, and we have shown how to define new patterns in MSL. As future work, we intend to extend the MSL pattern library with other common patterns of interacting MAPE loops, and the language itself to allow composition strategies of patterns instances in the same design.

Another direction that we intend to pursue is the evaluation of the usability and usefulness of the framework on a certain number of case studies, and to evaluate mapping towards other back-end frameworks to improve MSL features. Finally, we also plan to extend our framework in order to support an automatic backward compatibility between an MSL model and its ASM counterpart.

Acknowledgments

P. Arcaini is supported by ERATO HASUO Metamathematics for Systems Design Project (no. JPMJER1603), JST. Funding Reference number: 10.13039/501100009024 ERATO. R. Mirandola is partially supported by Linnaeus University, Sweden. E. Riccobene is supported by the European Union Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 691178.

References

- Abeywickrama, D.B., Hoch, N., Zambonelli, F., 2013. SimSOTA: engineering and simulating feedback loops for self-adaptive systems. In: Proc. of the Int. C* Conf. on Computer Science and Software Engineering. ACM, NY, USA, pp. 67–76.
- Alkhabbas, F., Spalazzese, R., Davidsson, P., 2017. Architecting emergent configurations in the internet of things. In: 2017 IEEE International Conference on Software Architecture, ICSA 2017, Gothenburg, Sweden, April 3–7, 2017. IEEE Computer Society, pp. 221–224. doi: 10.1109/ICSA.2017.37.
- Arcaini, P., Gargantini, A., Riccobene, E., 2010. AsmetaSMV: A Way to Link High-Level ASM Models to Low-Level NuSMV Specifications. In: Abstract State Machines, Alloy, B and Z. Springer, Berlin, Heidelberg, pp. 61–74.
- Arcaini, P., Gargantini, A., Riccobene, E., 2010. Automatic Review of Abstract State Machines by Meta Property Verification. In: Proceedings of the Second NASA Formal Methods Symposium (NFM 2010). In: NASA Conference Proceedings, NASA/CP-2010-216215. NASA, pp. 4–13.
- Arcaini, P., Gargantini, A., Riccobene, E., Scandurra, P., 2011. A model-driven process for engineering a toolset for a formal method. Softw. Pract. Exper. 41 (2), 155–166.
- Arcaini, P., Mirandola, R., Riccobene, E., Scandurra, P., 2019. A pattern-oriented design framework for self-adaptive software systems. In: 2019 IEEE International Conference on Software Architecture Workshops, ICSA Workshops 2019, Hamburg, Germany, March 25–29, 2019.
- Arcaini, P., Mirandola, R., Riccobene, E., Scandurra, P., 2018. A DSL for MAPE patterns representation in self-adapting systems. In: 12th European Conference on Software Architecture, ECSA 2018, Madrid, Spain, Sept. 24–28, 2018, Proceedings. Springer, pp. 3–19.
- Arcaini, P., Riccobene, E., Scandurra, P., 2017. Formal design and verification of self-adaptive systems with decentralized control. ACM Trans. Auton. Adapt. Syst. 11 (4), 25:1–25:35.
- Bonfanti, S., Gargantini, A., Mashkoor, A., 2018. AsmetaA: Animator for Abstract State Machines. In: Abstract State Machines, Alloy, B, TLA, VDM, and Z. Springer International Publishing, Cham, pp. 369–373.
- Börger, E., Raschke, A., 2018. Modeling Companion for Software Practitioners. Springer, Berlin, Heidelberg.
- Brun, Y., Di Marzo Serugendo, G., Gacek, C., Giese, H., Kienle, H.M., Litoiu, M., Müller, H.A., Pezzé, M., Shaw, M., 2009. Engineering self-adaptive systems through feedback loops. In: Software Engineering for Self-Adaptive Systems [outcome of a Dagstuhl Seminar], pp. 48–70.
- Calinescu, R., Weyns, D., Gerasimou, S., Iftikhar, M.U., Habli, I., Kelly, T., 2018. Engineering trustworthy self-adaptive software with dynamic assurance cases. IEEE Trans. Software Eng. 44 (11), 1039–1069.
- Carioni, A., Gargantini, A., Riccobene, E., Scandurra, P., 2008. A scenario-based validation language for ASMs. In: Proceedings of the 1st International

- Conference on Abstract State Machines, B and Z. Springer-Verlag, Berlin, Heidelberg, pp. 71–84.
- Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Di Marzo Serugendo, G., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H.M., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H.A., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., Whittle, J., 2009. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (Eds.), *Software engineering for self-adaptive systems: a research roadmap*. Springer Berlin Heidelberg, Berlin, Heidelberg.
- Cheng, S., Garlan, D., 2012. *Stitch: a language for architecture-based self-adaptation*. *J. Syst. Softw.* 85 (12), 2860–2875.
- Clark, T., van den Brand, M., Combemale, B., Rumpe, B., 2014. Conceptual model of the globalization for domain-specific languages. In: Cheng, B.H.C., Combemale, B., France, R.B., Jézéquel, J., Rumpe, B. (Eds.), *Globalizing Domain-Specific Languages – International Dagstuhl Seminar Dagstuhl Castle, Germany, October 5–10, 2014 Revised Papers*. Springer, pp. 7–20. doi:10.1007/978-3-319-26172-0_2.
- D'Angelo, M., Caporuscio, M., Napolitano, A., 2017. Model-driven engineering of decentralized control in cyber-physical systems. In: *2017 IEEE 2nd Int. Workshops on Foundations and Applications of Self* Systems (FAS*W)*, pp. 7–12.
- Darby, S.J., 2018. Smart technology in the home: time for more clarity. *Build. Res. Inf.* 46 (1), 140–147.
- De Lemos, R. et al., 2013. Software Engineering for Self-adaptive Systems: a Second Research Roadmap. In: *Software Engineering for Self-Adaptive Systems II*. Springer, pp. 1–32.
- Gargantini, A., Riccobene, E., Scandurra, P., 2008. A metamodel-based language and a simulation engine for abstract state machines. *J. Universal Comput. Sci.* 14 (12), 1949–1983.
- Gargantini, A., Riccobene, E., Scandurra, P., 2009. A semantic framework for metamodel-based languages. *Automated Software Engg.* 16 (3–4), 415–454.
- Garlan, D., Cheng, S., Huang, A., Schmerl, B., Steenkiste, P., 2004. Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer* 37 (10), 46–54.
- Harel, D., Rumpe, B., 2004. Meaningful modeling: what's the semantics of "semantics"? *Computer* 37 (10), 64–72.
- Hebig, R., Giese, H., Becker, B., 2010. Making control loops explicit when architecting self-adaptive systems. In: *Proceedings of the 2nd International Workshop on Self-organizing Architectures*. ACM, New York, NY, USA, pp. 21–28.
- Iftikhar, M.U., Weyns, D., 2017. Activforms: a runtime environment for architecture-based adaptation with guarantees. In: *2017 IEEE International Conference on Software Architecture Workshops, ICSA Workshops 2017*, Gothenburg, Sweden, April 5–7, 2017, pp. 278–281.
- Iglesias, D.G.D.L., Weyns, D., 2015. MAPE-K formal templates to rigorously design behaviors for self-adaptive systems. *ACM Trans. Auton. Adapt. Syst.* 10 (3), 15:1–15:31.
- Kautz, O., Roth, A., Rumpe, B., 2018. Achievements, failures, and the future of model-based software engineering. In: Gruhn, V., Striemer, R. (Eds.), *The Essence of Software Engineering*. Springer, pp. 221–236. doi:10.1007/978-3-319-73897-0_13.
- Kephart, J.O., Chess, D.M., 2003. The vision of autonomic computing. *IEEE Computer* 36 (1), 41–50.
- Kříkava, F., Collet, P., France, R.B., 2014. ACTRESS: Domain-specific Modeling of Self-adaptive Software Architectures. In: *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. ACM, NY, USA, pp. 391–398.
- Meliá, S., Cachero, C., Hermida, J.M., Aparicio, E., 2016. Comparison of a textual versus a graphical notation for the maintainability of MDE domain models: an empirical pilot study. *Softw. Qual. J.* 24 (3), 709–735.
- Microsoft, 2009. *Microsoft Application Architecture Guide*, second Microsoft Press.
- Mirandola, R., Riccobene, E., Scandurra, P., 2019. Self-accounting in architecture-based self-adaptation. In: *Proceedings of the 13th European Conference on Software Architecture - Volume 2*. ACM, New York, NY, USA, pp. 14–17. doi:10.1145/3344948.3344957.
- Runeson, P., Höst, M., 2009. Guidelines for conducting and reporting case study research in software engineering. *Empir. Softw. Eng.* 14 (2), 131–164. doi:10.1007/s10664-008-9102-8.
- Voelter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L.C.L., Visser, E., Wachsmuth, G., 2013. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. <http://www.dslbook.org>.
- The MSL language, 2018. The MSL language. <https://github.com/fimslab/msl>.
- Weyns, D., Iftikhar, M.U., de la Iglesia, D.G., Ahmad, T., 2012. A survey of formal methods in self-adaptive systems. In: *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering*. ACM, New York, NY, USA, pp. 67–79. doi:10.1145/2347583.2347592.
- Weyns, D., Schmerl, B.R., Grassi, V., Malek, S., Mirandola, R., Prehofer, C., Wuttke, J., Andersson, J., Giese, H., Göschka, K.M., 2013. *Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany*. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 76–107. *On Patterns for Decentralized Control in Self-Adaptive Systems*.
- Wohlin, C., Runeson, P., Hst, M., Ohlsson, M.C., Regnell, B., Wesslén, A., 2012. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated.

Paolo Arcaini is project associate professor in the ERATO MMSD project at the National Institute of Informatics, Japan. His main research interests are related to model-based testing, search-based testing, fault-based testing, and automatic repair. His home page is <http://group-mmm.org/~arcaini/>.

Raffaela Mirandola is associate professor in the Dipartimento di Elettronica, Informazione e Bioingegneria at Politecnico di Milano, Italy. Her research interests include the areas of performance and reliability modeling and analysis of software/hardware systems with special emphasis on: methods for the automatic generation of performance and reliability models for component-based and service-based systems, and methods to develop software that is dependable and can easily evolve, possibly self-adapting its behavior. Her home page is <http://home.deib.polimi.it/mirandola/>.

Elvina Riccobene is full professor at the University of Milan, Italy. Her research topics include formal methods, Abstract State Machines, integration between formal modelling and modeldriven engineering, model analyses techniques for software systems. Her home page is <https://homes.di.unimi.it/riccobene/>.

Patrizia Scandurra is associate professor at the University of Bergamo, Italy. Her research interests include integration of formal and semiformal modeling languages for software engineering; formal methods, and analysis (validation and verification) techniques for pervasive systems; MDE; model-driven methodologies, techniques, and tools for the design and analysis of embedded systems and systems on chip (SoC). She is a member of the Abstract State Machines (ASMs) formal method community. Her home page is <https://cs.unibg.it/scandurra/>.