A Formal Model of Metacontrol in Maude [⋆]

Juliane Päßler[1], Esther Aguado[2], Gustavo Rezende Silva[3], S. Lizeth Tapia Tarifa[1], Carlos Hernández Corbato[3], and Einar Broch Johnsen[1]

[1] University of Oslo, Oslo, Norway
`{julipas,sltarifa,einarj}@ifi.uio.no`
[2] Universidad Politécnica de Madrid, Madrid, Spain
`e.aguado@upm.es`
[3] Technical University of Delft, Delft, the Netherlands
`{g.rezendesilva,c.h.corbato}@tudelft.nl`

Abstract. Nowadays smart applications appear in domains spanning from commodity household applications to advanced underwater robotics. These smart applications require adaptation to dynamic environments, changing requirements and internal system errors. Metacontrol takes a systems of systems view on autonomous control systems and self-adaptation, by means of an additional layer of control that manipulates and combines the regular controllers. This paper develops a formal model of a Metacontrol architecture. We formalise this Metacontrol architecture in the context of an autonomous house heating application, enabling different controllers to be dynamically combined in order to meet user requirements to a better extent than the individual controllers in isolation. The formal model is developed in the Maude rewriting system, where we show results comparing different scenarios.

1 Introduction

There is an emerging development of "smart" applications in an increasing number of domains, ranging from commonplace IoT-based household applications to advanced underwater robotics. A key to this smartness is the applications' ability to flexibly adapt to variability in their operational conditions. Control theory and recent AI-based methods enable the development of different domain controllers that adapt to variability in the physical environment by combining sensor data and a diversity of models to determine the output of the system actuators, achieving great performance and a degree of system-level autonomy for different requirements. This is called first-order self-adaptation [15]. However, varying user requirements and internal system variability due to faults or emergent behaviour in systems that comprise multiple controllers pose additional second-order adaptation challenges [15]. Self-adaptive systems with multiple points of view and methodologies have been proposed from the software

[⋆] This work was supported by the European Union's Horizon 2020 Framework Programme through the MSCA network REMARO (Grant Agreement No 956200) and ROBOMINERS project (Grant Agreement No. 820971).

community to develop software systems that meet the previous adaptation requirements. These systems have been commonly conceptualised following the MAPE-K schema [6, 18] that stands for Monitor, Analyse, Plan, and Execute based on Knowledge.

Metacontrol is a layered framework for achieving autonomy, independently of the application and the system; i.e., it takes a very structured system of systems view on the self-adaptation problem. It adds an additional layer of control that manipulates and combines the domain controllers to fulfil different combinations of requirements or to ensure resilience by enabling the overall application to adapt to changes in the underlying system such as, e.g., broken devices. Thus, Metacontrol enables extensible self-adaptation for autonomous systems that can accommodate new user requirements or new functionality to an application, by addressing only second-order adaptation based on application-independent reasoning. For this purpose, its MAPE-K loop is driven by knowledge that adheres to TOMASys [13] (the *Teleological and Ontological Model of Autonomous Systems* metamodel), which aims to provide the necessary concepts for modelling the functional knowledge of autonomous systems [16].

In this paper, we formalise a Metacontrol architecture for a smart house, detailing an autonomous house heating application. This use case is inspired by the work of Arcaini et al. [2] on a model of concurrent MAPE-K controllers (but not of Metacontrol), using abstract state machines [5]. Our main focus here is on the logical structure of architectures based on Metacontrol; i.e., the model captures the input assumed to be available to the different layers of the control structure and the control decisions that the different layers make. We develop models of independent controllers for the house heating application and show that each of these is able to maintain some of the given user requirements, but not the overall combined requirements. These models are then extended with a Metacontrol layer that, by dynamically combining the different existing controllers, is able to meet the requirements to a greater extent.

The formalisation is realised in Maude [12], a tool to develop and analyse executable models in rewriting logic. Rewriting Logic is a flexible framework combining computation and logic in which systems can be modelled with low representational distance [19]. In this paper, we use simulations in Maude to analyse the performance of the controllers with respect to the user requirements in fairly predictable environments; in future work, we plan to expand our model to analyse the additional responsiveness that seems achievable by exploiting a Metacontrol layer in less predictable environments by means of model checking techniques.

The main contributions in this paper are:

- an executable formal model of a system of systems Metacontrol architecture developed in the Maude rewrite tool;
- a use case of autonomous layered control for a smart house heating application, which we use to explain and illustrate the formal model; and
- we show how the formal model can be used to analyse the developed metacontrol designs with respect to user requirements.

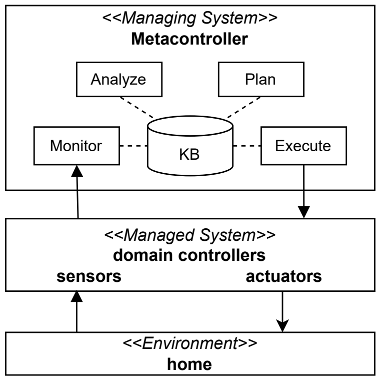

Fig. 1: The Metacontrol framework.

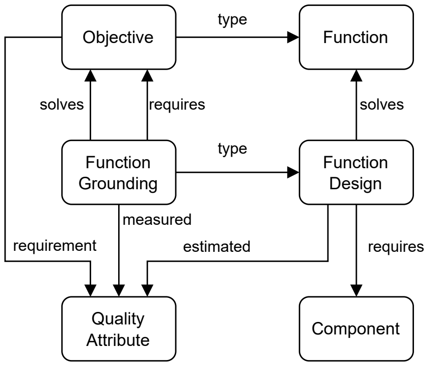

Fig. 2: Elements of TOMASys.

2 Background

2.1 Metacontrol

Metacontrol [13] gives systems the capability to perform self-adaptation in order to maintain their functionalities at an expected performance, in the presence of external disturbances, faults, and unexpected behaviour. Thus, Metacontrol enables an increased level of autonomy and reliability in a system.

Metacontrol is realised by adding an extra control layer that closes a feedback loop with the original system, monitoring, and adapting the original system through architectural reconfiguration when necessary. According to the MAPE-K model for self-adaptive systems, this results in the separation of the system into two distinct layers or subsystems: the *managing system*, whose main element is a reusable metacontroller, and the *managed system*, which is the original system.

For the metacontroller to be application independent, it exploits engineering knowledge at runtime. System engineers model relevant information about how the system works following the TOMASys metamodel [13], and this model forms the basis for the knowledge base (KB), which drives the Metacontrol MAPE-K adaptation loop. All steps interact with the KB, storing additional information, and retrieving system knowledge to facilitate decision making (see Figure 1).

The *monitor* step is responsible for measuring so-called *quality attributes* (discussed below) for the functionalities of the system. The *analyse* step is responsible for deciding whether the managed system needs to reconfigure. For this, it needs to decide whether the measured quality attributes meet the requirements of the current configuration of the managed system. When reconfiguration is needed, the *plan* step is responsible for selecting a new configuration for the managed system that potentially satisfies its requirements. The *execute* step is responsible for carrying out the reconfiguration of the managed system.

Figure 2 shows a high-level representation of the TOMASys metamodel for the Metacontrol functional layer, which is our focus in this paper. A *function* represents an abstract functionality of the system, e.g., controlling the temperature. A *function design* is an engineering design solution that solves a specific function, e.g., a specific controller to control the temperature. An *objective* is a concrete instance of a function that the system desires to achieve at runtime, e.g., maintaining the room temperature in the range 18–22°C. A *function grounding* is the runtime instantiation of a function design that is selected to solve a specific objective. A quality attribute (QA) is a measurable property of a system that is used to indicate how well the system satisfies its requirements, e.g., the amount of energy that the temperature control system consumes. In TOMASys there are *required*, *expected*, and *measured* QAs:

– *required QAs* capture quantitative requirements that a function should meet;
– *estimated QAs* are assumed for each function design as expected performance values with respect to the corresponding quantitative requirements; and
– *measured QAs* are associated to function groundings to capture the current performance with respect to the corresponding quantitative requirements.

2.2 Maude

Maude [12, 22] is a specification and analysis system based on rewriting logic (RL) [19]. While algebraic specification techniques [25] can be used to specify the static structure of the system and the relations between the data, RL extends an algebraic specification with transitions rules which capture the dynamic behaviour of a system. In a rewrite theory (Σ, E, R), Σ defines the (ground) terms *tr* as operators over sorts (which can be understood as types), E is a set of equations that define equivalences between terms in Σ, and R is a set of labelled rewrite rules.

Equational theories (Σ, E) are developed by first defining *sorts st* and *functions* from a (possibly empty) list *stlist* of sorts to a given sort *st*:

```
sort st .
op f : stlist ⟶ st .
```

Terms tr are then built from functions in a sort-correct manner, and *patterns t* are terms in which some functions are replaced by variables. *Conditional equations* in E, which express the equality of any terms that match given patterns if a given condition holds, are specified by

```
ceq t = t' if cond
```

where the matching substitution additionally applies to the condition. The condition can be a conjunction of equalities between patterns. Maude assumes that the equations form a terminating and confluent reduction system which is used to represent equivalence classes by canonical terms (i.e., all terms can be reduced to unique normal forms).

Rewrite theories (Σ, E, R) additionally specify *conditional rewrite rules* in R, which express that a term matching a given pattern can transition into another term if a condition holds, as follows:

Req.	Name	Description
R_1	*Morning water heating*	The water heater should be turned on in the morning.
R_2	*Minimise dispersion*	The system should avoid high dispersion states, i.e., window wide open and strong heating together.
R_3	*Comfortable temperature*	The smart house temperature shall always be between a range of comfortable temperatures.
R_4	*Air purity*	The heating system shall maintain a good air quality level.

Fig. 3: Requirements of the heating system. Here, the requirements R_3 and R_4 are quantitative system properties which are monitored by the quality attributes.

```
crl [label] : t ⇒ t′ if cond .
```

Here, the condition is a conjunction of rewrites and equalities that must hold for the rule to apply to a given term which matches t. Rewrite rules apply to equivalence classes of terms; i.e., Maude uses the equations to reduce terms to normal form in between rule applications. Therefore, when auxiliary functions are needed, these can be defined in equational logic and thus evaluated in between the state transitions [19].

When modelling executable systems, system components are typically modelled by terms of suitable sorts, organised hierarchically in modules, and the global state configuration is represented as a multiset of these terms [22]. An object in a given state can have the form $\langle \mathsf{Oid} : \mathsf{Class} \mid \mathsf{a}_1 : \mathsf{v}_1, \ldots, \mathsf{a}_n : \mathsf{v}_n \rangle$, where Class is the class name, Oid the object identifier, and a_i are attributes with corresponding values v_i. Given an initial configuration, the Maude tool supports simulation and breadth-first search through reachable states, and model checking of systems with a finite number of reachable states [12, 22].

3 The Smart House Use Case

Let us consider a smart house with an automatic heating system. This use case was originally presented in [2]. Here, we present a variation of the use case to motivate a self-adaptive heating system that includes various controllers that partially fulfil the requirements and a metacontroller that turns the different controllers on and off with the aim of improving the system behaviour. The house consists of a room with one window, one heater and one water heater. Its intended behaviour is specified through a set of requirements defined in Figure 3. The smart house system has various controllers that control three actuators, the heater, the water heater, and the window, and it has a temperature sensor (a thermometer), an air quality sensor, and a global clock to keep track of time.

Act.	Stat.	Temp.	Air Qu.
Heater	*VH*	1.23	-0.9
	FH	0.615	-0.45
	OFF	0	0
Water heater	*ON*	0,3	-0,125
	OFF	0	0
Window	*O*	-2.0	2.0
	HO	-1.0	1.0
	C	0	-0.125

Fig. 4: Assumed actuator effects on temperature and air quality

The heater can be set to very hot *VH*, fairly hot *FH* or *OFF*, the water heater to *ON* or *OFF*, and the window to open *O*, half open *HO*, or closed *C*. A thermometer measures the temperature as a float value, where a comfortable temperature ranges over values between (and including) 18 and 22 °C. An air quality sensor measures the air quality as a float value, where a good level is represented by a value ≥ 0. The clock ranges over integer values from 1 to 24, and changes in a round-robin manner to represent discrete time units as time advances during a 24-hour day. We define all time units between, and including, 6 and 12 to be *MORNING* and all other time units as *NOT MORNING*.

In Figure 4 we model the impact that the state of the actuators has on the temperature and air quality. For every time step, the total impact on the temperature and air quality is calculated as the sum of the effects of each actuator. The temperature of the smart house is also affected by the environment, e.g., the outside temperature, which varies depending on the time of the day. We further consider additional variability of the environment (see the additional online material).

3.1 The Controller Layer

The operation of the smart house system is based on controllers that receive readings from the temperature and air quality sensors as well as the clock value as input. Depending on the sensor readings and whether it is morning or not, the active controller selects a status for each actuator. The controller's goal is to fulfil some of the requirements displayed in Figure 3 by changing the state of the actuators. The smart house heating system has four controllers: a controller that prioritises temperature (comfort controller), a controller that prioritises air quality (eco controller), and two degraded controllers, which only act upon one of the heaters, using only two of the three actuators in the smart house.

The controllers have policies based on rules to decide upon the state of the actuators, depending on the inputs (temperature, air quality, and clock), and assuming that the actuators' effect on the temperature and air quality in the room are as the presented in Figure 4.

The comfort controller uses all actuators and has a strong preference for *comfort temperature*. Its control policy is defined to meet to a certain degree the requirements in Figure 3: (1) a policy stricter than R_3: reach and maintain the desired comfort temperature measurements as a priority, even if that means slightly worse air purity measurements, and (2) R_1: the water heater is always on in the morning.

The controller policy is depicted as a decision diagram in Figure 5. The nodes represent the temperature T, clock CLK, air quality AQ, and window W. The edges represent the status of each node. The leaves represent the desired state of the actuators, as described in Figure 4, in the order: heater, water heater, and window. Note that the controllers check whether the temperature is within 19.0°C and 21.0°C and whether the air quality is above 1, so they use different

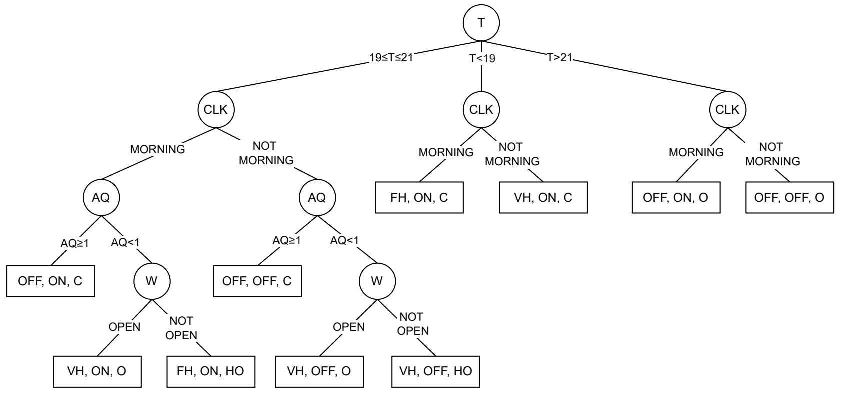

Fig. 5: Decision diagram for the comfort controller, where the leaf nodes indicate the controller's decision for the state of the heater, water heater and window.

values than the ones defined in the system requirements. This was done to create some buffer during which the controllers can react before the temperature and air quality pass the thresholds of 18.0°C and 22.0°C, respectively 0. Based on the decision tree, we can already observe that some requirements of Figure 3 are not met, as the controller prioritises a comfortable temperature. In particular, some rules violate R_2 since the controller uses the very hot heater state VH and the open window state O together.

The eco controller uses all actuators and has a strong preference for *air purity*. Its control policy is defined to meet to a certain degree the requirements in Figure 3: (1) a policy stricter than R_4: reach and maintain the desired air purity measurements as a priority, even if it means a slightly worse comfortable temperature measurements, (2) R_1: the water heater is always on in the morning, (3) an extension of R_2: if the temperature is high and it is morning, only reduce the temperature with the window half open to avoid losing heat too quickly.

Degraded controllers are defined for each heater that may break, so the degraded controllers can be used when that happens. *Degraded controller A* assumes that the heater is broken and therefore always *OFF*. *Degraded controller B* assumes that the water heater is broken and therefore always *OFF*.

Decision diagrams, similar to the one showed in Figure 5, are included for the remaining controllers in the online repository[1].

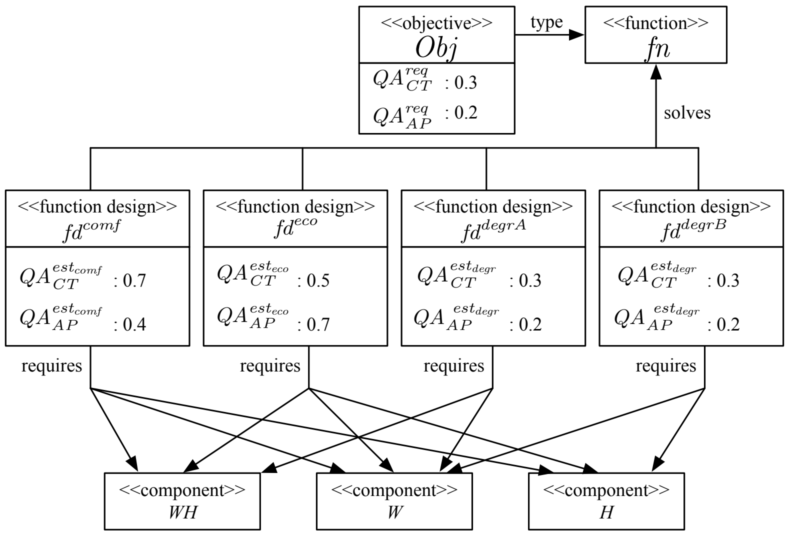

Fig. 6: Elements of the smart house TOMASys metamodel.

4 The Metacontrol Layer for the Smart House Use Case

In this section we present the Metacontrol layer for the smart house use case, modelling R_3 and R_4 in Figure 3 as an abstract functionality of the system that the metacontroller aims to maintain. The overall system includes all the controllers presented in Section 3.1. The comfort controller prioritises R_3 over R_4, while the policy of the eco controller does the opposite. In case a heater breaks, the degraded controllers can be used. The adaptation problem that the metacontroller solves is configuring the smart house application by selecting at each instant the best controller to perform both R_3 and R_4.

4.1 The TOMASys Metamodel

The quantitative requirements R_3 and R_4 are captured in the TOMASys metamodel, as explained in Section 2.1. Figure 6 provides an overview of the TOMASys model for this use case.

Function. Requirements R_3 and R_4 can be represented as a *function fn*: use a controller to keep the smart house temperature and air quality at a desired level. Observe that in the general case, the metacontroller can operate based on different functions that depend on different quality attributes at the same time.

[1] The full model of the smart house Metacontrol architecture, including all scenarios and an appendix, is available at **https://github.com/remaro-network/Maude_Metacontrol**.

Quality attributes. To quantify the extent to which the system satisfies the requirements R_3 and R_4, we use the *quality attributes* QA_{CT} (comfortable temperature) and QA_{AP} (air purity). These quality attributes occur as *required* quality attributes in the objective, as *estimated* quality attributes in the function designs, and as *measured* quality attributes in the function grounding, and will be further discussed in the respective paragraphs.

Objective. An objective is a concrete instance of the function *fn*. A desired range for the temperature and a good level for the air quality are given by the *objective Obj*: keep the smart house temperature between (and including) 18 and 22 °C and the air quality ≥ 0. To evaluate whether an objective has been fulfilled, the metacontroller uses minimum thresholds for each quality attribute, such thresholds are denoted as *required quality attributes* QA^{req}_{CT} and QA^{req}_{AP}. These values also reflect the priorities that are set in the system. The higher the value, the higher the priority. For the smart house use case, we want to prioritise comfortable temperature; therefore $QA^{req}_{CT} > QA^{req}_{AP}$, see Figure 6. We denote this objective as $Obj(QA^{req}_{CT}, QA^{req}_{AP})$. Observe that, in general, there is one objective for each function since the objective is an instance of a function.

Function design. We denote each function design in Figure 6 as

$$fd^{id}(QA^{est_{id}}_{CT}, QA^{est_{id}}_{AP}, Cmp_{id}) \ ,$$

where

- *id* is one of the identifiers of the controllers,
- QA^{est}_{CT} and QA^{est}_{AP} are *estimated* quality attribute values for comfortable temperature and air purity, and
- $Cmp \subseteq \{heater, waterheater, window\}$ are *required* components.

Function grounding. If a controller is selected, we *ground* its function design. This instance is then called *function grounding*. In the function grounding, we store the *measured* quality attributes QA^{meas}_{CT} and QA^{meas}_{AP} that reflect the degree to which the quality attributes are fulfilled at the moment.

If one of the actuators fails or the measured quality attribute values are lower than the required ones, then the function grounding will be marked as in error, which will trigger reconfiguration in the metacontroller. This includes grounding a (new) function design.

Quality attribute values. Figure 6 includes the required and estimated values for the quality attributes used in the use case. The function design fd^{comf} captures the goal of the comfort controller: to prioritise a comfortable temperature in the room and then, if the temperature is good enough, try to maintain a reasonable air quality, thus $QA^{est_{comf}}_{CT} > QA^{est_{comf}}_{AP}$. The function design fd^{eco} focuses on a good air quality, therefore $QA^{est_{eco}}_{CT} < QA^{est_{eco}}_{AP}$.

For simplicity all degraded controllers have the same estimated values in our model, which are assumed to be lower than the estimated values for the other controllers, and prioritise room temperature.

All the above elements are used by the metacontroller to decide when to switch between the different controllers, so that the system does its best to meet both requirements R_3 and R_4.

4.2 Metacontrol Operation

We now detail the operation of the metacontroller, following the MAPE-K loop, where the knowledge base of the MAPE-K loop includes all the elements and relationships among the elements defined in Section 4.1.

Monitor. During the monitor step, the metacontroller uses some collected logs in the system to calculate the current measured values of the quality attributes. In particular, we assume that the heating system has a log for the temperature values in the room $T = t_0, t_1, \ldots, t_n$ and a log for the air quality values in the room $A = a_0, a_1, \ldots, a_n$, where $t_i, a_i \in \mathbb{Q}$ for all $i = 0, \ldots, n$. They were retrieved from the sensors every time step $0, \ldots, n$, where n is the current time step. Let $N \in \mathbb{N}$ be the size of the time window that we want to consider for the computation of the current values of the quality attributes. We now decide whether a temperature t_i is considered to be a comfortable temperature. Let $CT : \mathbb{Q} \longrightarrow \{0, 1\}$ be defined as

$$CT(t_i) = \begin{cases} 1 & \text{if } 19.1 \leq t_i \leq 21.5, \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

The measured quality attribute QA_{CT}^{meas} for comfortable temperature is then given by the function $QA_{CT}^{meas} : \mathbb{N} \times \mathbb{Q}^n \longrightarrow [0, 1]$, defined as follows:

$$QA_{CT}^{meas}(N, T) = \frac{\sum_{i=(n-N)}^{n} CT(t_i)}{N}. \tag{2}$$

We compute the measured air purity as a function $AP : \mathbb{Q} \longrightarrow \{0, 1\}$, defined as:

$$AP(a_i) = \begin{cases} 1 & \text{if } a_i \geq 0.7, \\ 0 & \text{otherwise.} \end{cases} \tag{3}$$

The measured quality attribute QA_{AP}^{meas} for air purity is then given by a function $QA_{AP}^{meas} : \mathbb{N} \times \mathbb{Q}^n \longrightarrow [0, 1]$, defined as follows:

$$QA_{AP}^{meas}(N, A) = \frac{\sum_{i=(n-N)}^{n} AP(a_i)}{N}. \tag{4}$$

Observe that the thresholds for comfortable temperature and air purity in CT and AP differ from the system requirements in Section 3. This difference anticipates the adaptation that will be done the metacontroller. Finally, if $n < N$, we only consider the first n time steps in both QA_{CT}^{meas} and QA_{AP}^{meas}.

Analyse. During the analysis step, the metacontroller first checks whether one of the actuators is in error. If yes, then the current function grounding is marked to be in error. If all actuators work, the metacontroller compares the measured quality attribute values QA_{CT}^{meas} and QA_{AP}^{meas} with the required QA values in the objective $Obj(QA_{CT}^{req},\ QA_{AP}^{req})$. If at least one measured QA value is lower than the respective required QA value, then the current function grounding is marked to be in error. The QAs that have a lower measured than required value are marked as underachieved. For example, if $QA_{CT}^{meas} = 0.2$, then QA_{CT} is marked as underachieving since $QA_{CT}^{req} = 0.3$, see Figure 6.

Plan. If the metacontroller marked the function grounding to be in error, then during the planning step the metacontroller needs to select a new function design to ground. In the case where both quality attributes are underachieved, the objective $Obj(QA_{CT}^{req}, QA_{AP}^{req})$ prioritises the QA for comfortable temperature since $QA_{CT}^{req} > QA_{AP}^{req}$.

Assuming that the system is at time step n, we define $available_n$ as the set of function designs that are available at time step n, i.e., the set of function designs whose required components only contain components that are not broken. Furthermore, we define $poorQA_n$ as the set of QAs that are underachieved at time step n. The planning step in the metacontroller acts slightly different depending on two cases:

Case 1: *The metacontroller only needs to look at the required components to choose a function design.* In our particular scenario, this is the case in which either the heater or the water heater is failing.
In this case $available_n \subseteq \{fd^{degrA}, fd^{degrB}\}$. Since in our scenario only one of the heaters can be broken, $available_n$ will have exactly one element, which the metacontroller chooses to be grounded in the execute step.

Case 2: *The metacontroller needs to look into the QAs that are marked as underachieved.* In our particular scenario, this is the case in which all the actuators are working normally. Thus, $available_n$ consists of all function designs. In this case $poorQA_n \subseteq \{QA_{CT},\ QA_{AP}\}$. Observe that $poorQA_n$ contains at least one element. Let $QA_x \in poorQA_n$ be the QA with the highest priority in $poorQA_n$. The metacontroller searches for the function design in $available_n$ with the highest estimated value for QA_x, i.e., the function design with identifier i such that for all function designs in $available_n$ with identifier j it holds that $QA_x^{est_i} \geq QA_x^{est_j}$.

Execute. During the execute step, the chosen function design is grounded and the controller associated with this function design is activated, triggering execution in the controller layer as described in Section 3.1.

5 Modelling the Smart House Use Case in Maude

We now describe the Maude model of the metacontrolled smart house application, including all elements described in Sections 3 and 4. A multiset `Configuration`

```
1  op ⟨Scheduler | Status: _, RuleApplied: _ ⟩ : ScheduleComp Bool ⟶ Scheduler .
2  op ⟨ _ : Clock | Timesteps: _, Time: _, TempLog: _, AqLog: _ ⟩
3       : Oid Timesteps Time TVPList TVPList ⟶ Clock .
4  op ⟨ _ : Thermometer | Degrees: _ ⟩ : Oid Temperature ⟶ Thermometer .
5  op ⟨ _ : Airquality | Value: _ ⟩ : Oid AirqualityStatus ⟶ Airquality .
6  op ⟨ _ : Heater | _ ⟩ : Oid Attribute ⟶ Heater .
7  op ⟨ _ : Waterheater | _ ⟩ : Oid Attribute ⟶ Waterheater .
8  op ⟨ _ : Window | _ ⟩ : Oid Attribute ⟶ Window .
9  op ⟨ _ : ComfortController | Selected: _ ⟩ : Oid Selected ⟶ ComfortController .
10 op ⟨ _ : EcoController | Selected: _ ⟩ : Oid Selected ⟶ EcoController .
11 op ⟨ _ : DegradedContrA | Selected: _ ⟩ : Oid Selected ⟶ DegradedContrA .
12 op ⟨ _ : DegradedContrB | Selected: _ ⟩ : Oid Selected ⟶ DegradedContrB .
13 op ⟨Environment | Version: _ ⟩ : Nat ⟶ Environment .
```

Fig. 7: Selected objects of the smart house heating system, modelled in Maude.

is used to represent the components of the Metacontrol ecosystem, as suggested in Section 2.2. A Configuration contains all objects that are crucial for the model of the smart house, detailed in the sequel. Figures 7 and 11 give the syntax of a selection of these objects. The full model can be found in the online repository.

5.1 Model Dynamics

The following elements model the execution of the entire self-adaptive system, as described in Figure 1.

Scheduler is an object of sort Scheduler (see Figure 7, Line 1) that captures the MAPE-K loop and ensures that rules are applied in the correct order, according to Figure 8. It keeps track of which sets of rules can be applied, using the attribute Status, and schedules which components can apply rules. The attribute RuleApplied indicates whether one of the rules in the active set has already been applied. The scheduler object is present in all rules. A rule can only be applied if the scheduler object has the right value in the attribute Status, and if the attribute RuleApplied is false, e.g., see the rule in Figure 9. The scheduler makes the model deterministic. However, different configurations will trigger different rules inside the sets.

Time is modelled with an object of sort Clock (see Figure 7, Line 2) that captures the passage of time in the system, which works together with a rule for modelling how time advances. It has attributes

– Timesteps: a natural number representing the current time step in the system,
– Time: the current time, and
– TVPList: a list of (Timestep, Float)-pairs.

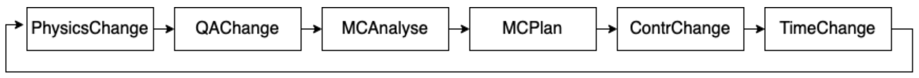

Fig. 8: The order in which the different set of rules are applied

After each time step, a pair representing the current time step and temperature (air quality) is added to the temperature log TempLog (respectively to the air quality log AqLog). As described in Section 3, we assume that one time step is one hour, so the time is computed as the current time steps modulo 24. Furthermore, a sort TimedConfiguration is introduced. It is used in certain rules to ensure that rules are applied to the full global state configuration and not only to subsets. If the scheduler indicates that the time should advance, then a rule that increases the Timesteps by one, computes the new Time and adds new pairs to TempLog and AqLog is applied.

5.2 The Smart House Model

In this section, we present the model of the managed system, as displayed in Figure 1 and described in Section 3.

Thermometer models the measurement of the current temperature in the room. An object of sort Thermometer (see Figure 7, Line 4) has the attribute Temperature that is a float value that reflects the current temperature in the smart house.

Air quality sensor models the measurement of the current air quality value in the room. An object of sort Airquality (see Figure 7, Line 5) has the attribute AirqualityStatus that is a float value that reflects the current air quality value in the smart house.

Actuators have an object identifier Oid and a multiset of Attributes (see Figure 7, Lines 6–8). Thus, the attributes that are necessary for an actuator are not explicitly specified. However, each actuator should have exactly one instance of the following attributes:

– Status: can be of sort HeaterStatus, WaterheaterStatus or WindowStatus, depending on the actuator, and represents the current status of the actuator;
– EffectTemp: the effect of the actuator on the temperature, specified as a float value;
– EffectAQ: the effect of the actuator on the air quality, specified as a float value; and
– Broken: indicates whether the actuator is broken or not.

See Figure 4 for further details about the effect of the actuators on the temperature and air quality.

```
crl [CContrTempOkAqOk] :
    ⟨ H : Heater | A ⟩ ⟨ WH : Waterheater | A1 ⟩ ⟨ W : Window | A2 ⟩
    ⟨ T : Thermometer | Degrees: DG ⟩ ⟨ AQ : Airquality | Value: AQS ⟩
    ⟨ C : Clock | Timesteps: TS, Time: TI, TempLog: TL, AqLog: AL ⟩
    ⟨ CC : ComfortController | Selected: true ⟩
    ⟨ Scheduler | Status: ContrChange, RuleApplied: false ⟩
    ⇒
    (msg hOff from CC to H)(msg whOff from CC to WH)(msg closed from CC to W)
    ⟨ H : Heater | A ⟩ ⟨ WH : Waterheater | A1 ⟩ ⟨ W : Window | A2 ⟩
    ⟨ T : Thermometer | Degrees: DG ⟩ ⟨ AQ : Airquality | Value: AQS ⟩
    ⟨ C : Clock | Timesteps: TS, Time: TI, TempLog: TL, AqLog: AL ⟩
    ⟨ CC : ComfortController | Selected: true ⟩
    ⟨ Scheduler | Status: ContrChange, RuleApplied: true ⟩
    if morning(TI) ==false /\ hot(DG) ==false /\ cold(DG) ==false
    /\ aqok(AQS) ==true .
```

Fig. 9: A rule for the comfort controller in Maude (coloured text highlights changed or new elements).

Physics. If the scheduler indicates that the temperature and air quality should be changed, then a rule is applied to change them according to the status of the actuators and the time, as specified in Section 3. We also implemented three different ways the environment can influence the temperature and air quality in the room, where two of them do not only depend on the time of the day, but also on the time step. The environment behaviour can be selected via the attribute Value of the object Environment (see Figure 7, Line 13).

5.3 The Controller Layer

The controllers, defined in Section 3.1, are modelled as objects; the attribute Selected indicates if a controller is currently active (see Figure 7, Lines 9–12).

The policies of the controllers are captured as rules. If a controller is active and the scheduler indicates that the controller should apply a rule, then the appropriate rule, which aims to change the status of the actuators, is applied. However, if an actuator is broken, its status should not be changed. We use a sort Msg for passing messages from a controller to an actuator, defined as **op** msg_from_to_ : ActuatorStatus Oid Oid ⟶ Msg . A broken actuator will not change its status when consuming the message. Otherwise it will change its status to the desired one.

The Maude model has rules that cover all the cases of the decision diagrams of the controllers. A sample rule for the comfort controller, given in Figure 9, captures a path in the decision diagram of Figure 5. This rule is applied when it is not morning and the temperature and air quality are OK. The pattern on the left hand side of the rule describes a configuration with the involved actuators, sensors, controllers. The scheduler has status ContrChange and a RuleApplied

```
rl [AnalyseDegA] :
    {⟨ H : Heater | Broken: yes, A ⟩
    ⟨ MC : Metacontroller | MetaLog: ML ⟩
    ⟨ Scheduler | Status: MCAnalyse, RuleApplied: false ⟩
    ⟨ ErrorPropagation | FgError: false, QaCtError: B, QaApError: B1,
        ActError: AIE ⟩
    CONF}
    ⇒
    {⟨ H : Heater | Broken: yes, A ⟩
    ⟨ MC : Metacontroller | MetaLog: ML ⟩
    ⟨ Scheduler | Status: MCAnalyse, RuleApplied: true ⟩
    ⟨ ErrorPropagation | FgError: true, QaCtError: false, QaApError: false,
        ActError: heater ⟩
    deselect(CONF)} .
```

Fig. 10: A rule for the metacontroller in Maude. The rule is applied when the heater is broken (coloured text highlights changed or new elements).

attribute with value false. The rule creates three messages and changes the attribute RuleApplied to true, which ensures that the controller only applies one rule per time step. The rule uses auxiliary functions; e.g., the function morning determines whether the current time is between 6 and 12 o'clock, cold, the functions cold and hot whether it is cold (i.e., less than 19.0°C) or hot (i.e., more than 21.0°C), and the function aqok whether the current air quality is above 1.0.

5.4 The Metacontrol Layer

We now present the model of the managing system, as shown in Figure 1 and described in Section 4.

Metacontroller is captured by an object of sort Metacontroller (see Figure 11, Line 1). The attribute MetaLog contains a list of pairs (Timestep, Value), where Value can be Eco, Comf, DegA, or DegB. At a time step i, the metacontroller might switch between controllers. If so, it adds a pair (i, C_i) to the MetaLog, where C_i represents the controller which has been activated.

A sample rule of the metacontroller in Maude is displayed in Figure 10. This rule is part of the set of rules for the analysis step. The rule is applied when the heater is broken to propagate the error and to signal in the Metacontrol layer that the system needs reconfiguration. Note that the curly brackets around the configuration enforce that this rule can only be applied to the configuration as a whole. The pattern on the left hand side of the rule describes a configuration with a broken heater, all the necessary components used by the metacontroller, as well as the whole configuration CONF which is not further specified. The scheduler has status MCAnalyse and a RuleApplied attribute with value false. The ErrorPropagation object has a FgError attribute with value false, QaCtError and

```
1 op ⟨ _ : Metacontroller | MetaLog : _ ⟩ : Oid MetaLog ⟶ Metacontroller .
2 op ⟨ RequiredQAs | requQaCT : _, requQaAP : _ ⟩ : Rat Rat ⟶ Objective .
3 op ⟨ _ : QaComfTemp | _ ⟩ : Oid QaAttributes ⟶ QaComfTemp .
4 op ⟨ _ : QaAirPurity | _ ⟩ : Oid QaAttributes ⟶ QaAirPurity .
5 op ⟨ _ : FDContr | _ ⟩ : Oid FDAttributes ⟶ FunctionDesign .
6 op ⟨ ErrorPropagation | FgError : _, QaCtError : _, QaApError : _, ActError : _ ⟩
7   : Bool Bool Bool ActuatorList ⟶ InError .
```

Fig. 11: Selected objects used by the metacontroller in Maude.

QaApError attributes with arbitrary boolean values, and an attribute ActError with an arbitrary list of actuators as value. The rule changes the RuleApplied attribute to true, which ensures that only one analyse rule can be applied every time step. Furthermore, it changes the attribute FgError to true because the function grounding is in error, the attributes QaCtError and QaApError to false, and the attribute ActError to heater because the heater is broken. Lastly, it uses the auxiliary function deselect to deselect the currently active controller.

Quality attributes can be *required*, *expected* and *measured* QAs, see Section 4.1. They occur in different parts of the model in Maude. The *required* QA values are captured in an object of sort Objective (see Figure 11, Line 2), where requQaCT and requQaAP are the required QA values for comfortable temperature and air purity, respectively. The *expected* QA values are captured in the function designs (described later). The *measured* QA values are captured in objects of sort QaComfTemp and QaAirPurity (see Figure 11, Lines 3–4), where QaAttributes is a multiset sort of attributes. Each quality attribute should include each of the following attributes exactly once:

- Consider: a natural number that reflects how many time steps should be considered for the computation of the quality attribute;
- Past: a list of Boolean values recording $CT(t_i)$ in QaComfTemp and $AP(a_i)$ in QaAirPurity, defined in Equations 1 and 3, for each time step i;
- Status: the current value of the quality attribute, computed according to Equation 2 for QaComfTemp and according to Equation 4 for QaAirPurity;
- QaComputed: a Boolean value that indicates whether the QA was computed in the current time step.

The rules that update the attribute Past and compute the new measured QA values are applied. when enabled by the scheduler.

Function design objects of sort FunctionDesign are defined for every controller Contr, where FDAttributes is a multiset sort of attributes (see Figure 11, Line 5). Function design objects include the following attributes:

- ConContr: the Oid of the controller connected to this function design;

Controllers	NoViol			IntViol		
	Tmp.	AirQu.	Total	Tmp.	AirQu.	Total
Comfort	0	48	48	0.00	16.45	16.45
Eco	29	0	29	15.19	0.00	15.19
Metacontroller	10	14	24	4.815	4.65	9.465

Fig. 12: Collected metrics from running Scenario 1 in Maude.

- ExpQaCT: the expected QA value for comfortable temperature;
- ExpQaAP: the expected air purity QA value; and
- RequActuators: a list of required actuators.

Function grounding is represented implicitly. Error propagation happens when certain actuators are in error and when quality attributes are under-achieved. This is captured with an object of sort InError (see Figure 11, Lines 6 and 7), where the Boolean values indicate whether the function grounding, the QA comfortable temperature, or the QA air purity are in error, and ActError is a (possibly empty) list of actuators that are in error.

An Analyse rule is applied before a Plan rule, when enabled by the scheduler. The rules follow the procedure described in Section 4.2.

6 Evaluating the Executable Model of the Smart House

This section reports on simulation experiments to evaluate the performance of the controllers and the impact of adding the Metacontrol framework to the system using the developed Maude model. The experiments consist of running the Maude model of the smart house for 200 time steps, measuring relevant metrics, and comparing how the system behaves with the comfort and eco controllers, and with the Metacontrol layer. The experiments were done for the following scenarios: (1) all actuators are working, (2) the heater breaks at time step 75, and (3) the water heater breaks at time step 75. The experiments were also repeated for two other variations of the environment, which gave similar results to the ones reported in the paper. Note that given an initial configuration, the model is deterministic due to the round-robin scheduling policy and the simplification of the variability in the environment. Therefore, one simulation is enough to capture the behaviour of the system in the different scenarios.

The requirements R_3 and R_4 reported in Figure 3 are instantiated as follows: We assume that a comfortable temperature is between 18 and 22 degrees and a good air quality level is above 0. The results obtained for Scenario 1 are summarised in Figures 12, 13, and 14, respectively, where we considered an initial temperature of 19°C and an initial air quality of 1.3. We collect metrics that measure the following: *NoViol* counts how many time steps the requirements R_3 and R_4 presented in Figure 3 are violated. If, for example, the air quality drops below 0 for 2 time steps, *NoViol* will be equal to 2. *IntViol* sums up the severity of the violation with respect to the thresholds. If for example the air quality is -0.5 in time step 1, -1 in time step 2, and 0 in time step 3, it would result in an intensity of violation of -1.5. The collected metrics are shown in Figure 12.

Figures 13 and 14 show that the comfort controller does not violate R_3, but violates R_4, and the eco controller violates R_3, but not R_4. When Metacontrol is used, both requirements are violated, but the number of times and the intensity of the violations are lower than when the comfort or eco controllers are used in isolation. Similar results were also obtained with the Metacontrol layer for Scenarios 2 and 3. Further experiments injected variability in the environment, where he metacontroller still performed better than the comfort- and eco controller in Scenario 1 and at least as good as the comfort controller in Scenarios 2 and 3. All the experiments can be reproduced following the instructions in the online repository.

The results show that for the smart house use case, the additional Metacontrol layer, as modelled in this work, allowed the smart house system to adapt and improve its performance in all the experiments reported in this section.

7 Related Work

Multiple works have addressed formal models of self-adaptive systems in the context of providing assurances [24]. ActivFORMS [17] is a self-adaptation approach that explicitly uses formal models at runtime in the form of timed automata to promote adaptation, thus ensuring that the properties verified at design time are guaranteed at runtime. Compared with Metacontrol, ActivFORMS requires to develop the entire system, i.e. first and second order self-adaptation, using formal methods, while Metacontrol allows to leverage existing domain controllers. ENTRUST [9] is a generic methodology to design reliable self-adaptive software and its associated assurance cases, based on developing verifiable models, such as the automata in ActivFORMS, to check if they satisfy certain properties, such as that the controllers are deadlock free. Our work relates to ActivFORMS and ENTRUST regarding the development and verification of the self-adaptive system models. Our aim is to use Maude to formally model Metacontrol architectures and use it to analyse their properties.

The analysis of self-adaptive systems has been addressed using various formal methods. For example, an abstract modelling framework based on automata is introduced in [3], several papers [10, 14, 21] use (a subclass of) Petri nets to model self-adaptive systems, and a domain specific language has been proposed to support compositional verification of self-adaptive cyber-physical systems [4].

In our work, we took inspiration from the work of Arcaina et al. [2] on modelling MAPE-K feedback loops with Abstract State Machines, which introduced the Smart House Case Study that we adapted in our paper. However, the controllers they considered in their work correspond to the managed system in our paper; i.e., they model the control level rather than the Metacontrol level considered in our work. Furthermore, they consider a scenario in which multiple controllers can be active at the same time, which differs from our work which

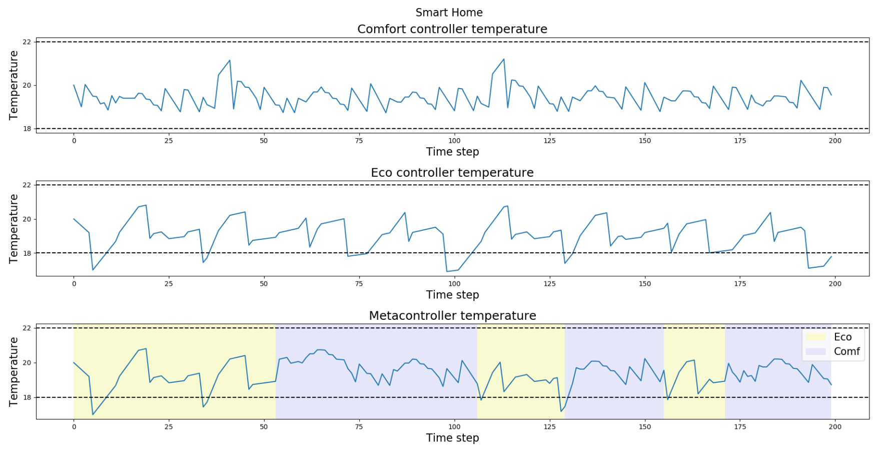

Fig. 13: The evolution of temperature for Scenario 1.

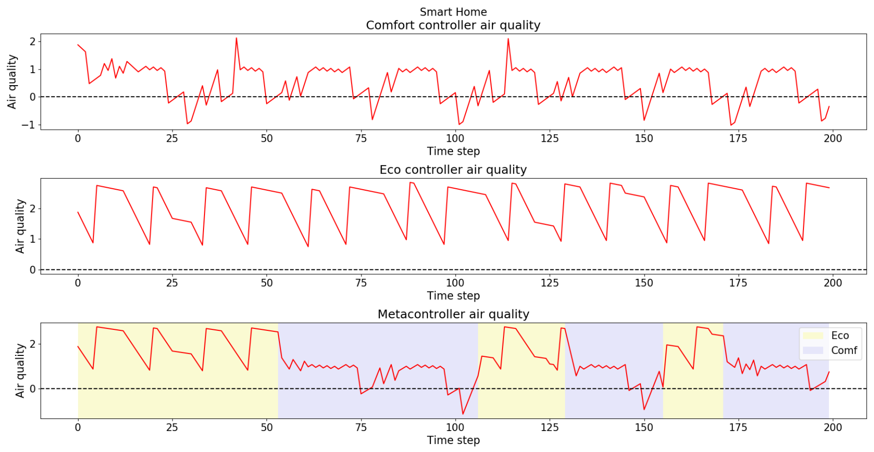

Fig. 14: The evolution of air quality for Scenario 1.

only considered the Metacontrol level activating and deactivating different controllers. Thus, they considered a different form of adaptation than in our work. Their work is further developed, for example, in [1].

Maude has previously been used to formalise and study reflection. In particular, rewriting logic allows a universal theory of reflection [11], which is also supported by Maude. Maude has also been used to formalise and study reflection in actor models using a "Russian dolls" architecture [20], especially in the context of reflective middleware [23]. Bruni et. al. [7] proposed a conceptual framework for self-adaptation that they realised in Maude [8]. Their work resembles ours in using a layered model of self-adaptation by means of MAPE-K feedback loops, but differs from ours in that they use a white box approach for implementing self-adaptation, whereas Metacontrol uses a black box approach.

8 Conclusion and Future Work

In this paper, we present the first executable formal model of a self-adaptive system based on Metacontrol. The use of Maude has allowed us to model a use case based on the smart house use case by Arcani et al. [2], including all the layers in such a self-adaptive system of systems, from the physics of the environment, via the first-order adaptation layer formed by the sensors, actuators and controllers, to the second-order adaptation layer defined by the Metacontrol architecture. The results presented in Section 6 for different scenarios of environment inputs and system failures validate our model of second-order self-adaptation with Metacontrol. They show an improved performance of the smart house system with respect to its requirements when the metacontroller performs architectural adaptation using the alternative controllers, compared to when the system's behaviour is limited to one of the individual controllers.

We highlight that the goal of this work is not to model the best controllers in the market, but to demonstrate that for scenarios where there are different controllers with specific advantages and limitations, Metacontrol can be used to capture and exploit this knowledge in order to improve the overall system performance. However, the potential to improve performance is deeply dependent on how well-designed the TOMASys metamodel is, especially the QAs.

In this paper, we have used simulations to analyse the benefits of Metacontrol in scenarios with fairly deterministic environments. In future work, we plan to use stronger analysis techniques, such as model checking and the exploration of what-if scenarios, to verify safety as well as liveness properties of the Metacontrol layer's abilities for self-adaptation in more unpredictable environments. For example, it would be interesting to verify properties related to the responsiveness and coverage; e.g., the Metacontrol layer's ability to always eventually regain correct behaviour, the maximal delay in this process, and whether the choice of active controllers will always fluctuate or eventually stabilise. Our long term goal is to analyse Metacontrol under assumptions about the managed system, and identify minimal requirements to the managed system such that the Metacontrol layer can guarantee desired overall correctness properties. A first step in this direction will be to generalise the formal model of the Metacontrol framework by replacing the executable models of the managed system and its environment by declarative specifications of their behaviour.

From the perspective of the design of the Metacontrol framework, our work on the Maude model has already provided insights into the current limitations of both the Metacontrol self-adaptation reasoning based on the TOMASys conceptual model and its execution model. In future work, we plan to extend the TOMASys metamodel to evaluate and prioritise conflicting requirements. Furthermore, we plan to explore the use of behavioural specifications of system components, as discussed above, to enrich the knowledge base of the Metacontrol architecture and integrate predictive analyses by means of formal models in the Metacontrol's self-adaptation layer.

References

1. Arcaini, P., Mirandola, R., Riccobene, E., Scandurra, P.: Msl: A pattern language for engineering self-adaptive systems. Journal of Systems and Software **164**, 110558 (Jun 2020). https://doi.org/10.1016/j.jss.2020.110558
2. Arcaini, P., Riccobene, E., Scandurra, P.: Formal Design and Verification of Self-Adaptive Systems with Decentralized Control. In: ACM Transactions on Autonomous and Adaptive Systems (2016)
3. Borda, A., Koutavas, V.: Self-adaptive automata. In: Proc. 6th Conference on Formal Methods in Software Engineering (FormaliSE 2018). pp. 64–73. ACM (Jun 2018). https://doi.org/10.1145/3193992.3194001
4. Borda, A., Pasquale, L., Koutavas, V., Nuseibeh, B.: Compositional Verification of Self-Adaptive Cyber-Physical Systems. In: Proc. 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2018). pp. 1–11 (May 2018)
5. Börger, E., Stärk, R.F.: Abstract State Machines. A Method for High-Level System Design and Analysis. Springer (2003)
6. Brun, Y., Serugendo, G.D.M., Gacek, C., Giese, H., Kienle, H.M., Litoiu, M., Müller, H.A., Pezzè, M., Shaw, M.: Engineering self-adaptive systems through feedback loops. In: Software Engineering for Self-Adaptive Systems. Lecture Notes in Computer Science, vol. 5525, pp. 48–70. Springer (2009)
7. Bruni, R., Corradini, A., Gadducci, F., Lluch-Lafuente, A., Vandin, A.: A conceptual framework for adaptation. In: de Lara, J., Zisman, A. (eds.) Proc. 15th International Conference on Fundamental Approaches to Software Engineering (FASE 2012). Lecture Notes in Computer Science, vol. 7212, pp. 240–254. Springer (2012). https://doi.org/10.1007/978-3-642-28872-2_17
8. Bruni, R., Corradini, A., Gadducci, F., Lluch Lafuente, A., Vandin, A.: Modelling and analyzing adaptive self-assembly strategies with Maude. Science of Computer Programming **99**, 75–94 (Mar 2015). https://doi.org/10.1016/j.scico.2013.11.043
9. Calinescu, R., Weyns, D., Gerasimou, S., Iftikhar, M.U., Habli, I., Kelly, T.: Engineering trustworthy self-adaptive software with dynamic assurance cases. IEEE Transactions on Software Engineering **44**(11), 1039–1069 (2017)
10. Camilli, M., Capra, L.: Formal specification and verification of decentralized self-adaptive systems using symmetric nets. Discrete Event Dynamic Systems **31**(4), 609–657 (Dec 2021). https://doi.org/10.1007/s10626-021-00343-3
11. Clavel, M.: Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications. CSLI Publications (2000)
12. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L. (eds.): All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic, Lecture Notes in Computer Science, vol. 4350. Springer (2007). https://doi.org/10.1007/978-3-540-71999-1
13. Corbato, C.H.: Model-based self-awareness patterns for autonomy. Ph.D. thesis, Universidad Politécnica de Madrid (2013)
14. Fakhir, M.I., Kazmi, S.A.R.: Formal Specification and Verification of Self-Adaptive Concurrent Systems. IEEE Access **6**, 34790–34803 (2018). https://doi.org/10.1109/ACCESS.2018.2849821
15. Garlan, D.: The unknown unknowns are not totally unknown. In: Proc. International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2021). pp. 264–265. IEEE (May 2021). https://doi.org/10.1109/SEAMS51251.2021.00047

16. Hernández, C., Bermejo-Alonso, J., Sanz, R.: A self-adaptation framework based on functional knowledge for augmented autonomy in robots. Integr. Comput. Aided Eng. **25**(2), 157–172 (2018)
17. Iftikhar, M.U., Weyns, D.: ActivFORMS: Active formal models for self-adaptation. In: Proc. 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2014). pp. 125–134. ACM (2014)
18. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer **36**(1), 41–50 (2003)
19. Meseguer, J.: Twenty years of rewriting logic. J. Log. Algebraic Methods Program. **81**(7-8), 721–781 (2012). https://doi.org/10.1016/j.jlap.2012.06.003
20. Meseguer, J., Talcott, C.L.: Semantic models for distributed object reflection. In: ECOOP. Lecture Notes in Computer Science, vol. 2374, pp. 1–36. Springer (2002)
21. Mian, N.A., Ahmad, F.: Modeling and Analysis of MAPE-K loop in Self Adaptive Systems using Petri Nets. International Journal of Computer Science and Network Security (IJCSNS) **17**, 6 (2017)
22. Ölveczky, P.C.: Designing Reliable Distributed Systems - A Formal Methods Approach Based on Executable Modeling in Maude. Springer (2017). https://doi.org/10.1007/978-1-4471-6687-0
23. Venkatasubramanian, N., Talcott, C., Agha, G.A.: A Formal Model for Reasoning About Adaptive QoS-Enabled Middleware. ACM Transactions on Software Engineering and Methodology (TOSEM) **13**(1), 86–147 (2004)
24. Weyns, D., Bencomo, N., Calinescu, R., Camara, J., Ghezzi, C., Grassi, V., Grunske, L., Inverardi, P., Jezequel, J.M., Malek, S., Mirandola, R., Mori, M., Tamburrelli, G.: Perpetual assurances for self-adaptive systems. In: de Lemos, R., Garlan, D., Ghezzi, C., Giese, H. (eds.) Software Engineering for Self-Adaptive Systems III. Assurances. pp. 31–63. Springer, Cham (2017)
25. Wirsing, M.: Algebraic specification. In: Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B), pp. 675–788. Elsevier and MIT Press (1990)