Computing (2023) 105:1849–1882
https://doi.org/10.1007/s00607-023-01167-9

SPECIAL ISSUE ARTICLE

Architecting decentralized control in large-scale self-adaptive systems

Jesper Andersson[1] · Mauro Caporuscio[1] · Mirko D'Angelo[2] · Annalisa Napolitano[3]

Received: 15 June 2022 / Accepted: 17 February 2023 / Published online: 9 March 2023
© The Author(s) 2023

Abstract

Architecting a self-adaptive system with decentralized control is challenging. Indeed, architects shall consider several different and interdependent design dimensions and devise multiple control loops to coordinate and timely perform the correct adaptations. To support this task, we propose DECOR, a reasoning framework for architecting and evaluating decentralized control. DECOR provides (i) multi-paradigm modeling support, (ii) a modeling environment for MAPE-K style decentralized control, and (iii) a co-simulation environment for simulating the decentralized control together with the managed system and estimating the quality attributes of interest. We apply the DECOR in three case studies: an intelligent transportation system, a smart power grid, and a cloud computing application. The studies demonstrate the framework's capabilities to support informed architectural decisions on decentralized control and adaptation strategies.

Keywords Reasoning framework · Model-based system engineering · Multi-paradigm modeling · Co-simulation

Mathematics Subject Classification 68-02

✉ Mauro Caporuscio
mauro.caporuscio@lnu.se

Jesper Andersson
jesper.andersson@lnu.se

Mirko D'Angelo
mirko.dangelo@ericsson.com

Annalisa Napolitano
annalisa.napolitano@imtlucca.it

1 Linnaeus University, Växjö, Sweden

2 Ericsson Research, Göteborg, Sweden

3 IMT School for Advanced Studies, Lucca, Italy

1 Introduction

Modern software systems are envisioned as large-scale distributed execution environments populated by myriads of pervasive real-world things, which collaborate to provide rich functionalities - e.g., smart homes, smart cities, intelligent traffic systems, and smart power grids. Since these applications operate under highly dynamic conditions, the traditional stability assumptions on systems' design are no longer valid. The dynamics introduce uncertainty, which may harm the system and lead to incomplete, inaccurate, and unreliable results [34]. Managing the run-time uncertainty is then crucial for the dependability of such systems.

Self-adaptation is widely considered a practical approach to deal with the uncertainty and dynamic of the environment [18]. Self-adaptive systems are conceptually organized as a *managed* system that implements the main functionality and a *managing* system that executes the adaptation logic through a control loop. The control is organized according to the well-established MAPE-K model [37], with Monitor (M), Analyze (A), Plan (P), and Execute (E) components, plus a Knowledge component (K) that maintains information the other components utilize.

When dealing with large-scale distributed systems, centralized control is hardly adequate to manage large-scale systems as several challenges must be tackled, e.g., maintaining consistent global knowledge, timely analyzing it, and routing all adaptation decisions through a centralized manager [14, 23, 25]. Hence, self-adaptation should be achieved through decentralized control, where multiple loops interact with each other to address run-time uncertainty [60].

Architecting decentralized control is challenging: architects shall consider several different and interdependent design dimensions and devise multiple control loops to coordinate and timely perform the correct adaptations. Typically, several architecture candidates exist to organize decentralized control. Different candidates entail different quality attributes (e.g., performance and cost), which architects should carefully evaluate when selecting a candidate for a given system.

In this work, we address the problem of adequately designing the decentralized control for a given managed system at the architecture level.To this end, architects should evaluate different architectures and determine the degree to which a decentralized control would provide the desired level of quality. In this context, reasoning frameworks [6] are used to guide architecture definition by predicting the extent to which a candidate architecture satisfies the quality requirements. Reasoning frameworks have been reported as a successful approach and are applied for assessing different types of quality attributes, e.g., modifiability [27], reusability [2], safety [36], and performance [43].

Therefore, our objective is to devise a reasoning framework that allows for (**O1**) modeling the decentralized control along with the managed system and (**O2**) evaluating how they affect each other. Towards this objective, the main contribution of this work is DECOR (DEcentralized COntrol Reasoning framework), a tool-supported reasoning framework satisfying the following requirements[1]:

[1] DECOR is available at the following link: https://github.com/mi-da/DECOR

R1.1 Multi-paradigm Modeling	The framework shall provide support for multi-paradigm modeling [33] and allow architects to specify the self-adaptive system as a multi-model composed of interacting models [10] (i.e., for the managing and managed systems), each using its formalism to specify entities and operations on a different spatial and temporal scale.
R1.2 Decentralized Control.	The framework shall support the design of decentralized control for large-scale distributed systems at the architecture level. The framework shall support multiple MAPE-K control loops and their constituents as first-class modeling elements [60].
R2 Evaluation	The framework shall provide support for evaluating the overall system (managing and managed systems), as well as focusing on attributes specific to the managed system under analysis.

The rationale for **R1.1** is to promote *reusability* and *separation of concerns* between managed and managing systems through multi-paradigm modeling; **R1.2** aims to tame *complex and distributed* systems through decentralized control; finally, **R2** aims to close the loop by providing support to evaluate the system quality.

Modeling control architectures has been largely addressed in Literature (e.g., [38, 40, 54]). However, the proposed approaches are agnostic to the underlying managed system and do not consider how the control impacts the managed system's behavior. Including the managed system in the modeling effort is the research gap this work addresses. We propose a *reasoning framework* with comprehensive support for multi-paradigm modeling and co-simulation of both the decentralized control and the managed system, which – to the best of our knowledge – is a novel contribution to the field of engineering self-adaptive systems. Indeed, DECOR leverages the model-based systems engineering paradigm [41] and promotes the systematic use of models as first-class elements in engineering decentralized self-adaptive systems. On the one hand, the reasoning framework allows analyzing the quality of a control loop with respect to system requirements. On the other hand, the model-based system engineering paradigm reduces the development complexity through a generative approach – i.e., generating different models, among which selecting the proper one [46].

DECOR has been developed by leveraging model-driven software engineering techniques [11]. In particular, we exploit a metamodel-centric design for (i) specifying a MAPE-K Control Modeling Language, (ii) defining a graphical modeling environment, and (iii) transforming the modeled MAPE-K Control into a simulation model. In order to demonstrate the framework's capabilities, we used DECOR in three different application scenarios: an intelligent transportation system, a smart power grid, and a cloud computing system. We use these scenarios to demonstrate how DECOR allows software architects to evaluate different Control architectures and make informed

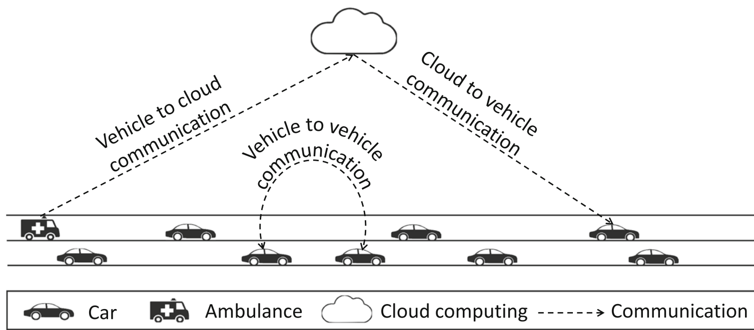


Fig. 1 Intelligent transportation system

decisions on decentralized control and adaptation strategies to meet the system requirements.

The remainder of the article is organized as follows. Section 2 introduces a running example illustrating the type of problems we intend to tackle. Section 3 discusses related work and describes the research gap. Section 4 formalizes the core elements of the proposed reasoning framework, whereas Sects. 5 and 6 provide details about DECOR implementation. Section 7 demonstrates how to use DECOR, and Section 8 discusses obtained results and threats to validity. Finally, Sect. 9 presents conclusions and hints for future work.

2 Running example

In this section, we describe a scenario serving as a running example throughout the paper to illustrate concepts and usage of DECOR.

An Intelligent Transportation System (ITS) is a transportation infrastructure enriched with control logic to optimize the overall quality and fulfill particular requirements, such as a situation-oriented goal. Figure 1 illustrates a 2-lane road section including *normal vehicles* (e.g., cars), *special vehicles* (e.g., ambulances), a wireless communication network, and a remote *cloud* computing infrastructure. The entities communicate with each other by exchanging data (see dashed lines) through different communication media (e.g., 5G network) and communication strategies (e.g., vehicle-to-vehicle, or vehicle-to-cloud).

The situation-oriented goal for the system is to *"minimize the average travel time of ambulances crossing the road section"*. That is, when an ambulance enters a congested road, the cars should prioritize their transit so that the average travel time for the ambulance shall be less than 50 seconds.

Different architectures might be employed to organize the control for such a system. For example, vehicles can interact with a centralized cloud computing infrastructure, which is in charge of planning the adaptation actions for the whole system. Alternatively, the vehicles can interact with each other in a fully decentralized peer-to-peer (P2P) fashion and exchange information to free a lane for the upcoming ambulance.

However, as different architectures entail different advantages and disadvantages (e.g., performance, and cost), they should be carefully evaluated to make sensible decisions, e.g., how should the control be organized? and how does the control, directly and indirectly, impact the system quality?

3 Related work

Implementation approaches for self-adaptive systems are categorized according to the type of integration between the control logic and the managed system [45]. In *internal approaches*, the control logic is intertwined with the managed resources, whereas *external approaches* separate the control logic from the managed system by reconciling them via sensors and actuators. This makes external approaches well-suited for implementing decentralized self-adaptive systems, as they promote separation of concerns and modularity, which lead to scalability and reusability [45]. Nevertheless, establishing paths from design space to system implementation, via architectural patterns, frameworks, and middleware is still recognized as one of the main research gaps for self-adaptive systems [12]. To this end, during the last few years, different *external approaches* have been proposed. Table 1 classifies the related literature concerning requirements **R1** and **R2**.

Abbas et al. [2] propose an engineering methodology called Autonomic Software Product Lines engineering (ASPLe). ASPLe provides developers with process support to implement product lines of self-adaptive systems with reuse at the managing system level. This work extends on ASPLe with dedicated support for architectural reasoning of decentralized self-adaptation in multi-domain systems.

The RAINBOW framework [31] makes use of architectural models to reason about the system's behavior and allows the explicit specification of reusable adaptation strategies for multiple concerns in a centralized self-adaptive system. In particular, RAINBOW aims to monitor the system's runtime properties, evaluate violations of the architectural model, and perform adaptation if a problem occurs. Differently from RAINBOW, we aim to support architects with a multi-paradigm modeling framework for designing and evaluating both the managing and managed system.

The FESAS framework [39] focuses on the reusability of the components defining the managing system architecture. On the other hand, we aim to investigate how different control patterns affect the managing architecture and the system's requirements.

Malek et al. [40] propose an architecture-based framework that supports engineering mobile software systems. The proposed framework offers complete life-cycle engineering support. However, the scope of [40] is restricted to mobile software systems, whereas we aim at embracing the more general class of large-scale self-adaptive systems. Moreover, the control design space of this work does not embrace the principle of MAPE-K components as first-class modeling abstractions for self-adaptive systems. Multi-paradigm modeling support is also not provided.

FUSION [29] aims at analyzing and self-tuning the adaptive behavior of a system in the presence of unanticipated changes. The focus of FUSION is on learning the impact of adaptation decisions on the system's goals at runtime for automatic online

Table 1 Related work classification

	Related work										Previous work	
	[2]	[31]	[39]	[40]	[29]	[9]	[38]	[3]	[54]	[57]	[22]	[24]
R1.1											✓	✓
R1.2			✓					✓		✓	✓	✓
R2	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓

tuning of the adaptation logic. On the other hand, we aim to provide architects with a general framework for explicitly designing decentralized control.

Bolchini et al. [9] propose a framework for specifying the characterizing elements for which the system will exhibit self-adaptive properties. Their model offers rigorous support for the identification and management of the aspects that characterize context and self-awareness. However, this framework does not support the design of the system's decentralized self-adaptation.

Kit et al. [38] advocate using component-based abstractions to engineer self-adaptive systems. The proposed framework allows for the simulation of real-time deployments by evaluating the system behavior under different network configurations and settings. However, this work does not provide mechanisms for designing decentralized control.

Arcaini et al. [3] propose a formal framework to specify self-adaptive systems. The framework allows for describing complex MAPE-K loops according to patterns and exploits validation and verification techniques for assuring the correctness of components' interactions. However, the investigation of timed adaptation is not possible. On the other hand, we leverage analytical theories that make it possible to evaluate these properties.

EUREMA [54] is a model-driven engineering approach providing a domain-specific modeling language and a runtime interpreter for feedback loops-based adaptations. DEUREMA [57] extends the former approach by supporting distributed feedback-loops coordination. Both works provide a modeling language that supports the explicit design of feedback loops. However, multi-paradigm modeling is not provided.

The related work survey indicates a focus on modeling the managing system architecture. In this respect, the approaches are agnostic to the managed system and provide interfaces to connect a given managed system. Hence, referring to Table 1, any analyzed related work does not provide multi-paradigm modeling support. Moreover, besides [40], no related work supports the evaluation of capabilities during architectural reasoning.

Further, Table 1 also reports how our previous work [22] [24] position with respect to the stated objectives. Even though both papers address requirements **R1** and **R2**, they do not present the DECOR reasoning framework as a whole nor relate the proposed approach with respect to state of the art. Specifically, in [22] we present the design of the multi-paradigm modeling and co-simulation environments, whereas [24] is devoted to the tool developed by leveraging [22]. With respect to these papers, the main additions and extensions presented herein are: (i) a specification of DECOR according to the reference architecture for reasoning frameworks defined in [6], (ii)

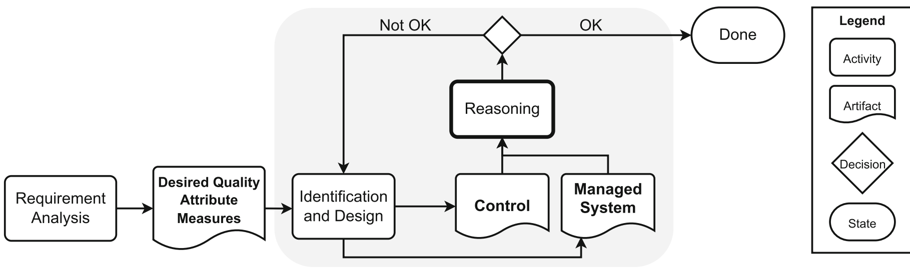


Fig. 2 Iterative reasoning process

a straightforward connection, specified in EMF [49], between the DECOR modeling language and FORMS, a well know reference model for the formal specification of distributed self-adaptive systems [59], and (iii) extensive experimentation demonstrating the applicability of DECOR in different settings.

4 Decentralized control reasoning framework

As introduced above, it is common to decentralize the control and deploy multiple coordinated adaptation elements in large-scale distributed systems. To this end, given a specific managed system, the proper architecture for organizing the decentralized control is identified through an iterative reasoning process (see Fig. 2) used by architects to make informed architectural decisions on the system under investigation. Such decisions might lead to potentially cutting out or exploring in detail those architectures that show bad or good levels of quality, respectively.

As depicted in Fig. 2, the process starts with the usual *Requirement Analysis*, which elicits the set of *Desired Quality Attribute Measures* for the to-be system. Then, *Identification and Design* is devoted to modeling the given *Managed System* and a candidate *Control*. Then, in the *Reasoning* activity the self-adaptive system (i.e., the managed system and the candidate control) is evaluated by assessing it with respect to the *Desired Quality Attribute Measures*. If the system does not satisfy the desired quality, the process feeds back into the *Identification and Design* activity by creating a design-evaluate loop (see Fig. 2). When the candidate architecture satisfies the desired quality, the design of the self-adaptive system is completed. Indeed, the *Reasoning* activity should consider two different concerns: (i) the quality of the control, and (ii) the impact of adaptations on the quality of the managed system itself. To facilitate the decision process, we need a systematic and integrated approach to perform the *Reasoning* and evaluate the control and its impact on the managed system.

Reasoning frameworks allow for evaluating quality attributes of the architecture under development through the use of well-defined theories. According to [6], a Reasoning Framework shall instantiate the following elements:

1. *Problem Description*: the set of quality attributes that can be calculated by the Reasoning Framework (e.g., latency, response time).

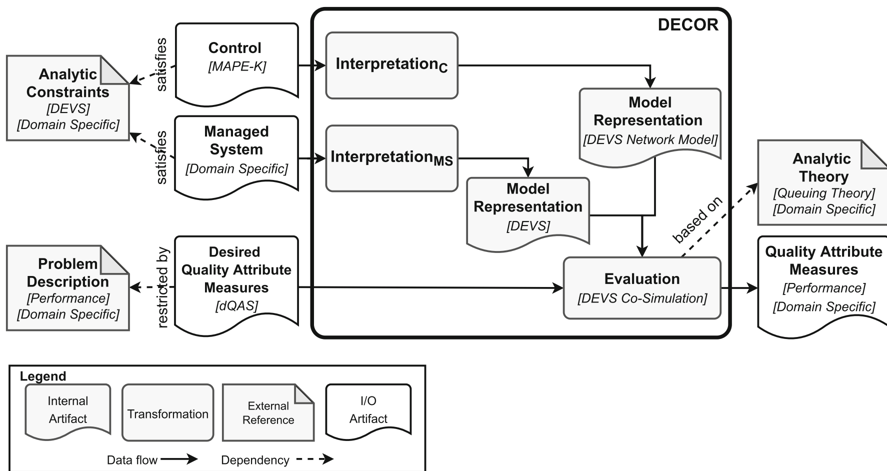


Fig. 3 DECOR approach

2. *Analytic Theory*: the theoretical foundations on which analyses are based (e.g., queuing theory).
3. *Analytic Constraints*: the set of constraints imposed on the analytic theory to restrict the design space (e.g., execution time).
4. *Model Representation*: the system model in a form suitable for use with the evaluation procedure (e.g., queues).
5. *Interpretation*: the procedure used to generate *model representations* from the architectural descriptions (e.g., from UML to queues).
6. *Evaluation Procedure*: the methods used to calculate quality measures from the model representation (e.g., analytical solution, simulation).

To this end, DECOR combines established analytic theories and model-driven engineering practices [11] to implement such key elements (see gray boxes in Fig. 3) and support the *Reasoning* activity.

Problem description Problem Description the reasoning framework is used. In particular, DECOR focuses on quality *performance* measures (e.g., adaptation time, scalability, etc..) at the *Control* level, as well as different *domain-specific quality measures* at the *Managed System* level. To this end, DECOR leverages the *extended Architectural Reasoning Framework* (eARF) [2] to specify the *Desired Quality Attribute Measures* as domain Quality Attribute Scenarios (dQAS). In particular, eARF focuses on requirements and design with explicit support for variability specification and modeling supports. To this end, eARF provides dQAS, a template for specifying the requirements of self-adaptive systems. The dQAS extends the well-known quality attribute scenario (QAS) [5] with three adaptation-specific elements: namely *Variants*, *Valid Configurations*, and *Fragment Constraints* (see description in Table 2).

Indeed, dQAS allows for specifying how a system should behave in specific situations.

For example, in the introduced ITS scenario, the quality requirement is *“When an ambulance enters the managed road section, the cars should prioritize its transit so*

Table 2 dQAS elements

Source	The origin of a Stimulus.
Stimulus	A condition that triggers self-adaptation.
Artifact	The affected parts of a system.
Environment	Operating environment under which a Stimulus arrives.
Response	How the system responds to the Stimulus.
Response Measure	How the Response is monitored and measured.
Variants	Different forms of a self-adaption property.
Valid Configurations	How the Variants can be combined.
Fragment Constraints	Constraints on the selection of dQAS elements' fragments.

Table 3 Desired quality attribute measure for the ITS scenario

Source	software (sub)system
Stimulus	state update: ambulances entering the road section
Artifact	The control subsystem
Environment	Runtime operating environment with any work-load
Response	Prioritize the transit of ambulances
Response measure	Ambulance travel time $<$ 50 seconds
Variants	centralized control and Decentralized control
Valid configurations	Not specified
Fragment constraints	Not specified

that the average travel time of the ambulance shall be less than 50 seconds", and the corresponding *Desired Quality Attribute Measure*, specified as dQAS, is shown in Table 3.

Analytic theory (and Constraints) Analytic Theory represents the foundations of the reasoning framework, as it enables the systematic analysis of the desired quality attribute measures. Indeed, it provides architects with concrete values for assessing and comparing different adaptations, as well as their impact on the managed system. Given the heterogeneity of the systems under consideration (e.g., traffics systems, smart grids, and pure software systems), DECOR employs a multi-paradigm *Analytic Theory*. Specifically, since *Queuing Theory* is widely recognized as a powerful and versatile tool for evaluating and predicting system performance [7], it is employed at the Control level for assessing its quality performance measures, e.g., *adaptation time*, *response time*, *latency*. On the other hand, other theories, suitable for the addressed domain and the specific quality measures of interest, are used at the Managed System level. For example, ITS makes use of traffic theory [52], whereas the Power Grid system makes use of electrical theory [55].

Employing *Queuing theory* at the Control level imposes some *Analytic Constraints* on the type of control that can be analyzed. Indeed, DECOR can be used to analyze and evaluate only Control defined as a discrete-event system (DEVS) [62]. On the

other hand, the managed system can be either discrete or continuous, depending on the specific domain addressed.

Model Representation Usually, model representation is relative to the analytic theory employed in the reasoning framework. To accommodate and evaluate a multi-paradigm *Analytic Theory*, DECOR employs DEVS [62] as ground *Model Representation*. DEVS is a popular formalism for modeling complex dynamic systems using a discrete-event abstraction. The main advantages of DEVS are the ability to model both discrete and continuous and hybrid systems, the support for modular composition, and the simulation-based analysis capability. Indeed, different DEVS models can be easily composed and simulated. While the Managed System is represented in DEVS, the Control is represented as a *DEVS Network Model*. In particular, MAPE-K components are modeled as nodes of a peer-to-peer network, which interact with each other via communication links. This allows for modeling the MAPE-K architecture as network topology, specifying how the interactions are carried on (e.g., point-to-point, point-to-multipoint, synchronous, asynchronous, etc.), and assessing the MAPE-K architecture in terms of network performance (e.g., throughput, latency, congestion, etc.).

Interpretation Interpretation refers to the mapping procedure converting the architectural model into the analysis model. Indeed, Interpretation is in charge of generating a *model representation* suitable for executing the *evaluation procedure* (i.e., a DEVS model) from an architecture description input to the reasoning framework [6]. Referring to Fig. 3, DECOR takes as input a model of the Control specified in terms of MAKE-K components, and a model of the Managed System specified according to a suitable domain-specific modeling language. For example, the ITS scenario is modeled in MovSim [51], the Smart Power Grid in Modelica [30], and the Cloud Computing system in CloudSim [13]. Indeed, the interpretation is a model transformation [11] from the specific input modeling language to DEVS. In particular, $Interpretation_{MS}$ is a model-to-model transformation from a Domain Specific modeling language to DEVS, whereas $Interpretation_{C}$ is a model-to-model transformation from *MAPE-K Modeling Language* (Sect. 5) to *DEVS Network Model* (Sect. 6.1).

Evaluation procedure The Evaluation Procedure is the application of the analytic theory to calculate specific quality attribute measures. Evaluation techniques might be formal, semi-formal, or simulation-based [44]. Formal methods (e.g., model checking, Petri-net, timed automata) are largely used in the literature for evaluating self-adaptive systems [3, 35, 58]. Despite their inherent ability to provide guarantees, these techniques do not scale well to large-scale systems. The effective use of formal methods requires reducing the computational problem by various techniques like model decomposition/composition or simplifying assumptions. However, even with simplifying assumptions and decomposition, the resultant analytic models are often not mathematically tractable. Therefore, a viable alternative for evaluating large-scale systems is through simulation. In particular, DECOR estimates the *Desired Quality Attributes Measures* by employing a co-simulation engine to simultaneously execute the DEVS model representing the Managed System and the DEVS Network Model representing the Control together. This enables for observing how the two models interact with each other and for evaluating how the Control affects the Managed System behavior (Sect. 6.2).

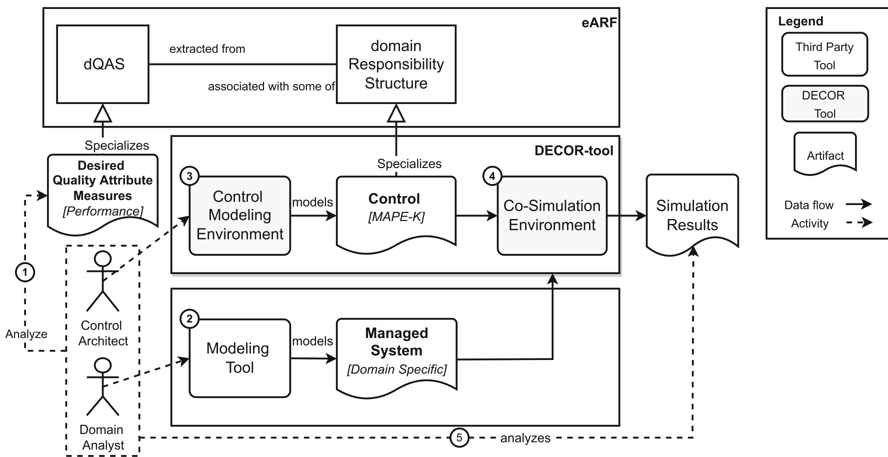


Fig. 4 Tool-supported reasoning

4.1 The DECOR tool-supported reasoning

In order to facilitate the reasoning activity and close the loop in the Reasoning Process (see Fig. 2), DECOR provides supporting tools for designing the *Control* and performing the evaluation through co-simulation. To this end, Fig. 4 shows the tool-supported process and depicts how stakeholders make use of DECOR core functionality, i.e., *Control Modeling Environment* (CME) and *Co-Simulation Environment* (CoSE). Tools are detailed in Sect. 6.

Specifically, *Control Architect* and *Domain Analyst* analyze the *Desired Quality Attribute Measures* and map them to *Control* (see ① in Fig. 4). Such a mapping is established through the *extended Architectural Reasoning Framework* (eARF) methodology [2], which revolves around four elements (see Fig. 5): namely dQAS, dRS, Architectural Tactics, and Architectural Patterns. As already discussed above, *dQAS* specifies the domain requirements for the self-adaptive system and describes how the system may react (adapt) to internal or external stimuli. *Domain responsibility structure* (dSR) models the decisions regarding architectural components, how they are structured, and the responsibilities assigned to each component. *Architectural Tactics* is a widely used architectural approach [4] to outline design decisions that influence the achievement of a given quality attribute response. In general, the specific architectural tactic to be employed is determined according to the given quality requirements. However, in this paper, we focus on *MAPE-K Tactic* (see Fig. 5). *Architectural Patterns* are structural schemas for organizing software systems, which provide a set of predefined subsystems, specify their responsibilities, and include rules and guidelines for implementing the software architecture. In general, an architecture pattern packages several tactics to realize one or more quality attributes. As DECOR focuses on *MAPE-K Tactic*, then we only consider MAPE-K Patterns [60]. The mapping from dQAS to dRS is performed by means of the Responsibility-Driven Design approach [61]. In particular, the responsibilities are extracted from dQAS and modeled as MAPE-K components in

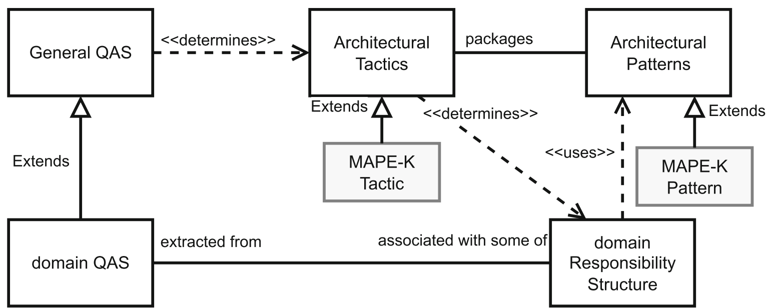


Fig. 5 extended Architectural reasoning framework [1]

the *Control* architecture. MAPE-K patterns are essential to this activity as they provide valuable knowledge to identify design alternatives (responsibilities and structure).

Once the mapping is established, the *Domain Analyst* (e.g., a traffic engineer in the ITS scenario) models the managed system functionalities by using a *Modeling Tool* (see ② in Fig. 4) suitable for the specific addressed domain (e.g., a micro-simulation traffic model [48]). The *Managed System* model is then imported in DECOR so that the observable and controllable elements (e.g., entities, relationships, properties, measures) of the managed system are automatically made available in CME. For example, referring to the ITS scenario, for each modeled vehicle (i.e., ambulances and cars), we can observe and control the position and speed.

Once the *Managed System* model is imported in CME, the *Control Architect* starts modeling the control structure to realize the adaptation logic (see ③) by mapping MAPE-K components to the observable and controllable elements of interest, according to the responsibilities extracted from the dQAS. Still referring to the ITS scenario, we *Monitor* the ambulance position, *Analyze* the road conditions (e.g., number of cars in the road section), and adapt the cars' position so that the ambulance's transit is prioritized.

When the managing and managed systems are ready, they can be evaluated in CoSE (see ④). DECOR provides for different evaluation strategies: the control and the managed system models can be simulated either separately to verify their properties in isolation (or together) to evaluate the overall system behavior. Based on the analysis of simulation results (see ⑤), architects may initiate a new iteration to improve the managing and managed systems model.

In the ITS scenario, DECOR allows for investigating the quality attributes concerning both the managed system (e.g., the average speed of vehicles) and the managing system (e.g., adaptation time, and the number of exchanged messages). These quality attributes highly depend on the specific MAPE-K control adopted, as different patterns can entail different results.

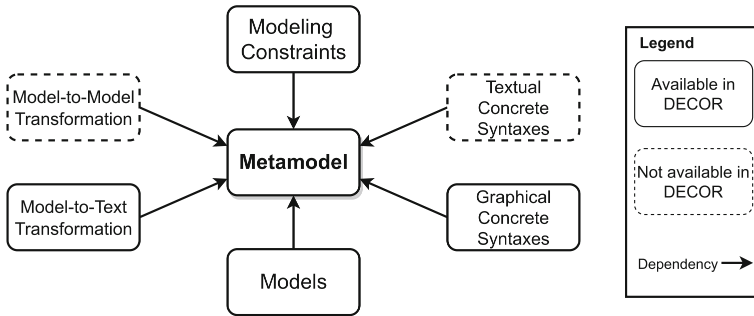


Fig. 6 Metamodel-centric language design [11]

5 MAPE-K modeling language

DECOR makes use of the MAPE-K tactic, which is widely recognized to be effective in realizing self-adaptation [37]. More specifically, DECOR relies on decentralized MAPE-K control, which allows for structuring and coordinating multiple interacting MAPE-K components (see Fig. 5). In [60], the authors introduce a simple notation for describing interactive MAPE-K loops. DECOR formalizes such a notation and provides a modeling language for specifying MAPE-K control loops.

To this end, the *MAPE-K Modeling Language* (MAPE- ML) is defined as a metamodel-centric language [11], where all aspects of the language are defined on the basis of the metamodel (see Fig. 6). In particular, the MAPE- ML metamodel is specified by using the Eclipse Modeling Framework (EMF) [49], which provides the metamodeling language *Ecore*. Ecore is based on a subset of UML class diagrams for describing structural aspects and is tailored to Java for implementation purposes (see Sect. 6). Specifically, Ecore is used to formalize the identified modeling concepts by modeling the abstract syntax of MAPE- ML. In contrast, modeling constraints are formalized by means of OCL [56], a declarative language for describing rules applying to UML models.

The output of this process is the MAPE- ML Metamodel depicted in Fig. 8. It is worth noticing that the metamodel takes root in FORMS [59], a well know and widely accepted reference model for the formal specification of distributed self-adaptive systems. This allows MAPE- ML to inherit and use concepts already defined in FORMS. FORMS consists of several modeling elements that specify the key aspects of a self-adaptive software system and the set of relationships guiding its composition. However, FORMS does not detail how the MAPE-K components should coordinate to perform the adaptation.

To this end, Figure 7 shows how the MAPE- ML Metamodel builds on FORMS and extends it to provide a concrete perspective on the coordination aspects concerning the MAPE-K components that realize the decentralized control. That is, MAPE- ML Metamodel allows for specifying how to structure and coordinate multiple MAPE-K components that interact with each other to achieve the adaptation goal.

FORMS defines a *Coordination Mechanism* as the rules governing the interactions among the participating computations [59]. *MAPE-K Control* extends such a concept

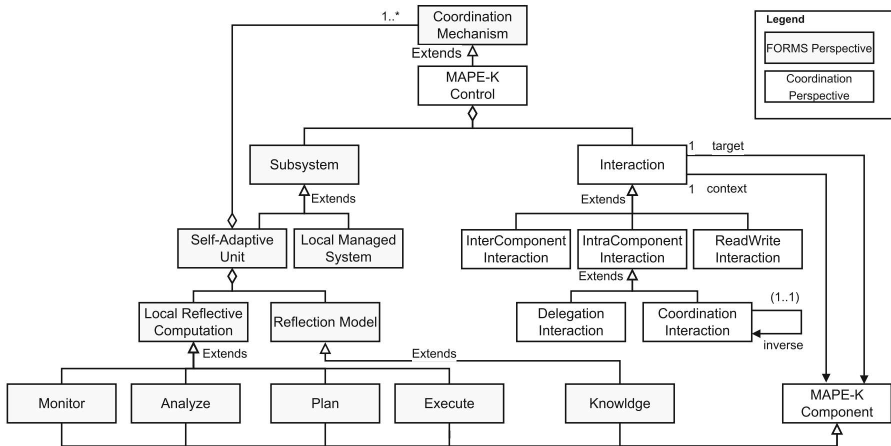


Fig. 7 Relation between FORMS and the MAPE- ML Metamodel

by concretizing it into a set of architectural elements to be instantiated. In particular, *MAPE-K Control* is composed of a set of architectural elements, namely *Subsystem* and *Interaction* (see Fig. 8). These elements define, respectively, the components and the connectors of the decentralized control.

Subsystem specializes in *Self-Adaptive Unit* and *Local Managed System*. *Local Managed System* models a managed system and specifies the set of *attributes* representing its observable/controllable aspects. *Self-Adaptive Unit* models the managing system in terms of five different types of *MAPE-K component*s, namely *Monitor*, *Analyze*, *Plan*, *Execute*, and *Knowledge*. Note that *Monitor* and *Execute* components can possibly be associated with a *Self-Adaptive Unit* to define complex hierarchical architectures, where a MAPE-K loop is controlling other MAPE-K loops. *Monitor* checks the *Subsystem* that is managed by the self-adaptive unit and stores collected data into the *Knowledge*. *Analyze* assesses the collected data in the *Knowledge* to determine whether *Subsystem* is satisfying its requirements. If the requirements are not satisfied, *Plan* determines the actions needed to adapt *Subsystem* and mitigate the observed problem. Finally, the *Execute* component changes the *Subystem*. It is whort noticing that MAPE- ML Metamodel allows for defining the architecture realizing the control (i.e., components and connectors) but not the behavior of the MAPE-K components, which is application specific (see Sect. 6.1).

Interaction models the communication between two *MAPE-K Component*s. The abstract *Interaction* defines a *context* and a *target*. The former indicates the component initiating the interaction, whereas the latter indicates the destination component. Following the notation used in [60], *Interaction* specializes in *InterComponentInteraction*, *IntraComponentInteraction*, and *ReadWriteInteraction*. In turn, *IntraComponentInteraction* is further specialized by *DelegationInteranction* and *CoordinationInteranction*. Specifically, *InterComponentInteraction* models the interactions between different types of MAPE-K components, whereas *IntraComponentInteraction* models the interactions between MAPE-K components of the same type, *CoordinationInteraction* models a bi-directional interaction used when two MAPE-K components have to

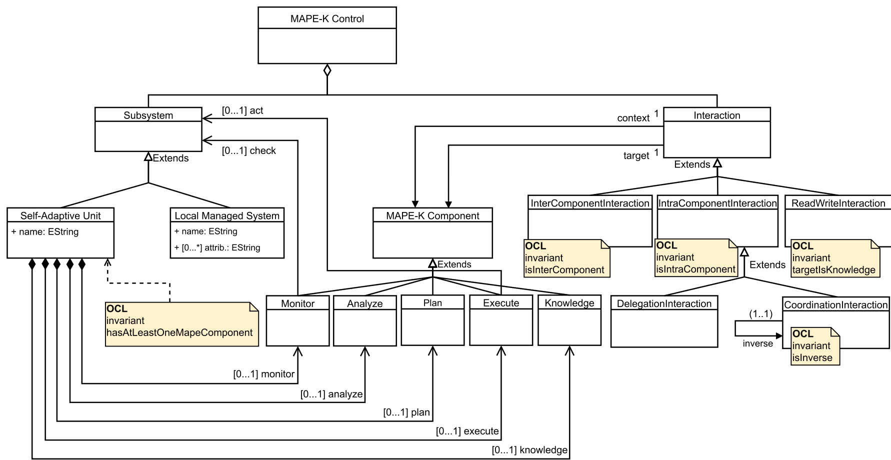


Fig. 8 Ecore metamodel for the MAPE-K modeling language

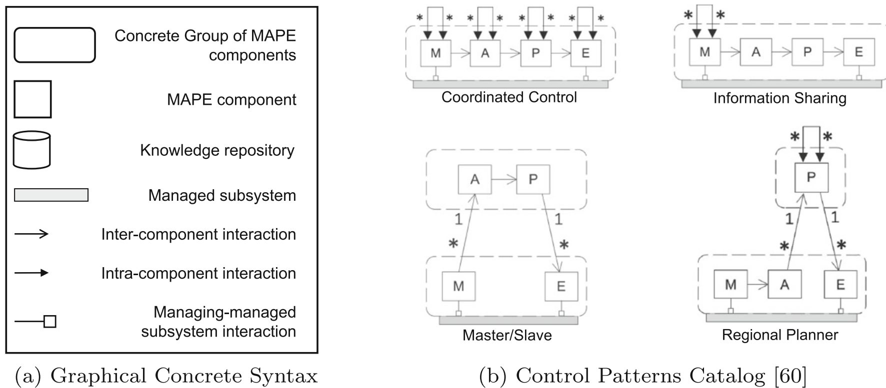
coordinate each other, *DelegationInteranction* models a mono-directional interaction, and *ReadWriteInteraction* represents the interaction between a MAPE component and a *Knowledge*.

As introduced above, to express constraints on the models, we make use of the OCL. Specifically (see Fig. 8), the MAPE- ML Metamodel defines five OCL rules or, using the OCL terminology, invariants. For a model to be valid, all its OCL invariants must hold[2]. For example, Listing 1 shows the OCL invariant *hasAtLeastOneMapeCompenent* (see Fig. 8), which specifies that a *Self-Adaptive Unit* must contain at least one *MAPE-K Component*. The invariant retrieves the MAPE-K components of the *Self-Adaptive Unit* (e.g., *self.monitor* in line 2), and checks whether at least one of them is not `null`.

Listing 1 invariant hasAtLeastOneMapeKComponent

```
invariant hasAtLeastOneMapeKComponent:
    self.monitor <> null or
    self.analyze <> null or
    self.plan <> null or
    self.execute <> null or
    self.knowledge <> null;
```

[2] The interested reader can find the complete formal specification of the OCL rules at the following link: https://github.com/mi-da/DECOR



(a) Graphical Concrete Syntax (b) Control Patterns Catalog [60]

Fig. 9 Modeling MAPE-K components in CME

6 DECOR supporting tools

As introduced above, DECOR provides two tools for supporting the reasoning activity, namely *Control Modeling Environment* (Sect. 6.1) and *Co-Simulation Environment* (Sect. 6.2).

6.1 Control modeling environment

The Control Modeling Environment (CME) is based on MAPE- ML and allows for architecting the control as a set of architectural elements, each implementing a specific MAPE-K component. The MAPE- ML graphical concrete syntax (depicted in Fig. 9a) is specified according to the notation introduced in [60], and allows for easy and effective modeling of MAPE-K control. CME allows users to architect control by specifying their own set of MAPE-K components and interactions or, as an alternative, by leveraging the ready-to-use control patterns [60] available in the *Control Pattern Catalog* (see Fig. 9b). For example, the *Master/Slave* pattern organizes into a hierarchical relationship where one (centralized) *Master* component is responsible for the analysis (A) and planning (P). In contrast, one or many *Slave* components are responsible for monitoring (M) and execution (E). On the other hand, the *Information Sharing* pattern organizes into a peer-to-peer relationship restricting the inter-component interactions to monitor (M) components only. Note that, the *Control Pattern Catalog* can be easily extended by defining new patterns. It is worth noting that the Control Patterns, as defined in [60], intentionally exclude the knowledge component to simplify the design and avoid dependencies of the knowledge element on system domains and underlying infrastructure. However, whether needed, CME allows architects to instantiate the knowledge component as it best suits the specific domain or infrastructure. CME is implemented as an Eclipse plugin and is based on the Eclipse Graphical Modeling Project[3], which provides proper mechanisms for developing model-driven development tools (see Fig. 10).

[3] http://www.eclipse.org/modeling/gmp/

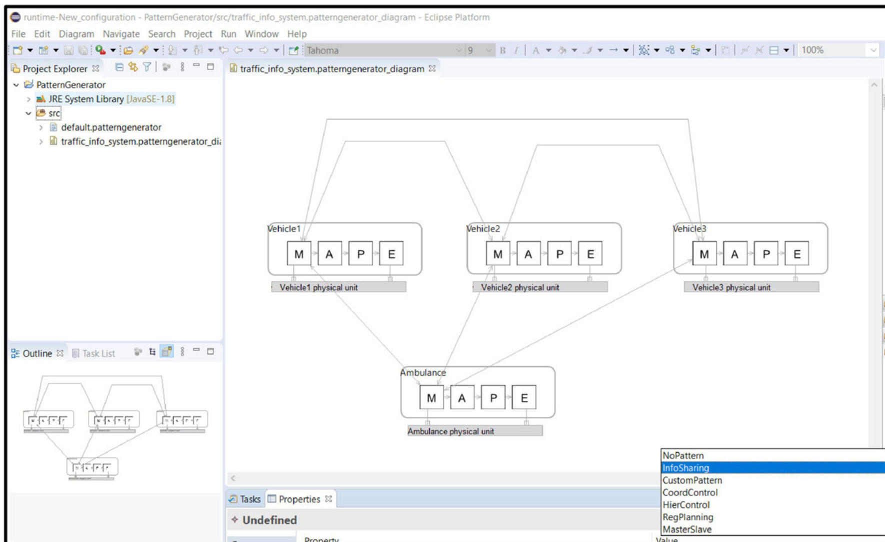


Fig. 10 DECOR Control modeling environment

As mentioned in Sect. 4 the *Managed System* model should be imported in DECOR. To this end, CME provides functionality for importing third-party models as a Functional Mock-up Unit (FMU) via Functional Mock-up Interface (FMI) [8], a tool-independent standard for the exchange of dynamic models and co-simulation. When the FMU is imported, the modeled entities are made available in CME so that they can be used as *Managed Subsystems* and their properties can be observed/manipulated. For example, in the ITS scenario, the FMU includes the road section and the vehicles, each with its own properties (e.g., position and speed).

Once the FMU entities are available in CME, architects model the MAPE-K control (by either instantiating one of the patterns in the catalog or making a customized one) and map the MAPE components to the underlying *Managed Subsystems* to be monitored and controlled. For example, in the IST scenario, the MAPE-K control can be implemented according to two different control patterns, namely the *Master/Slave* pattern and the *Information Sharing* pattern. In the *Master/Slave* pattern, the *Master* can be deployed in the Cloud, whereas the *M* and *E* components are mapped to all vehicle entities. This pattern models a centralized control in charge of analyzing (*A*) the data monitored by all the entities and planning (*P*) the adaptation strategy for each vehicle, which will execute the adaptation. On the other hand, in the *Information Sharing Pattern* shown in Fig. 10, all vehicles are running all the MAPE components. Such a pattern models a scenario where the information concerning the current status of each vehicle is shared among all the MAPE-K loops, which in turn locally analyze the situation, and plan/execute their adaptation in isolation. When the MAPE-K control is finalized, CME validates whether it is compliant with the MAPE- ML Metamodel and/or with one of the Control Patterns in the Catalog. The correctness of the model is validated through OCL. For example, Listing 2 shows one of the OCL rules defining the Information Sharing pattern shown in Fig. 10[2]. In particular, according to the

pattern definition (see Fig. 9b), the invariant *intraComponentInteractionIsMCoord* states that the only *IntraComponentInteraction* allowed is the one occurring between *MapeKComponents* of type *Monitor* (line 6 and 7).

Listing 2 invariant intraComponentInteractionIsMCoord

```
invariant intraComponentInteractionIsMCoord:
 if(type = PatternType::InfoSharing)
   then self.interactions -> select(oclIsKindOf(IntraComponentInteraction))
    -> select(oclAsType(Interaction).targetAndContextNotNull)
    -> reject(oclIsKindOf(Coordination) and
    oclAsType(Interaction).context?.oclIsKindOf(Monitor) and
    oclAsType(Interaction).target?.oclIsKindOf(Monitor)) -> size() = 0
   else true
 endif;
```

Valid models can finally be evaluated in the Co-Simulation Environment. To this end, CME exploits the EMF parser to perform a *Model-to-Text Transformation* and generate a *DEVS Network Model* of the MAPE-K control (see Sect. 4). Indeed, the modeled MAPE-K components are mapped into *protocol classes* according to the PeerSim specifications [42]. PeerSim is a discrete-events simulator designed to efficiently simulate large-scale peer-to-peer networks. The Model-to-Text Transformation consists of a set of transformation rules executed while parsing the MAPE-K model to generate a set of PeerSim-compliant Java classes. Once the Java classes are generated, the end-user can edit them by adding the specific behavior needed for implementing the adaptation policy. Note that CME currently does not provide architects with the ability to specify the behavior of the MAPE components (e.g., how to analyze data, adapt, etc.). While Mape- ML allows for defining the components and connectors to realize the control, the end user should specify the specific behavior of modeled components. Future extensions of Decor will allow for formally specifying the behavior of MAPE components (e.g., [35]).

6.2 Co-simulation environment

The Decor Co-Simulation Environment (CoSE) includes a *Co-Simulation Engine* for co-simulating the *Managed System* model and the *MAPE-K Control* modeled in CME. CoSE implements different simulation strategies by enabling the estimation of different quality attribute measures: the control and the managed system models can be simulated separately to verify their properties in isolation or together to evaluate the overall system behavior.

Co-simulation refers to the ability to simultaneously execute multiple simulation models by allowing information exchange among them [16]. To this end, Decor leverages MECSYCO (Multi-agent Environment for Complex SYstem CO-simulation) [15], a multi-models co-simulation platform. MECSYCO is based on the Agents & Artifacts for Multi-Modeling paradigm [47] and is implemented as a multi-agent system, where each model corresponds to an *Agent*, and the information exchange between the models correspond to the interactions between the agents. Indeed, each interaction

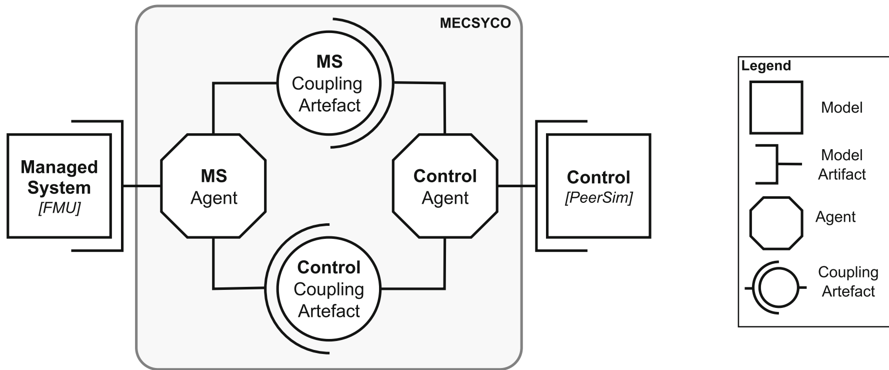


Fig. 11 Co-simulation environment (modeled with MECSYCO notation [47])

between agents is reified by an *Coupling Artifact* that holds a buffer used by agents to exchange input/output events. An agent handles its associated model as a DEVS atomic model through the *Model Artifact*, which implements the DEVS simulation protocol and acts as a DEVS wrapper for the model.

Figure 11 shows how the MECSYCO platform is exploited to implement CoSE and co-simulate the Control model and the Managed System model. In particular, the FMU representation of the Managed System is managed through an FMU Artifact that exposes a DEVS view of the FMU [17]. On the other hand, the PeerSim representation of the Control is managed through the PeerSim Artifact that exposes a DEVS view of the PeerSim model. To this end, we re-engineered PeerSim by implementing the following set of functions, which are needed by MECSYCO for handling the co-simulation[4]:

- $init()$ initializes PeerSim by setting the parameters, and the initial state of the model
- $processExternalEvent(e, t, k)$ processes the external input event e at simulation time t in the kth input port of PeerSim model.
- $processInternalEvent(t)$ processes the internal event of the PeerSim model scheduled at simulation time t.
- $getOutputEvent(k)$ returns the external output event e at the kth output port of PeerSim model.
- $getNextInternalEventTime()$ returns the time t of the earliest scheduled internal event of the PeerSim model.

Once the Managed System and Control are exposed as DEVS model, their simulation is coordinated by MECSYCO via the *Managed System* (*MS*) *Agent* and *Control Agent*, which interact with each other by mean of *MS Coupling Artifact* and *Control Coupling Artifact*, respectively.

[4] For further details, refer to the MECSYCO website: http://mecsyco.com/

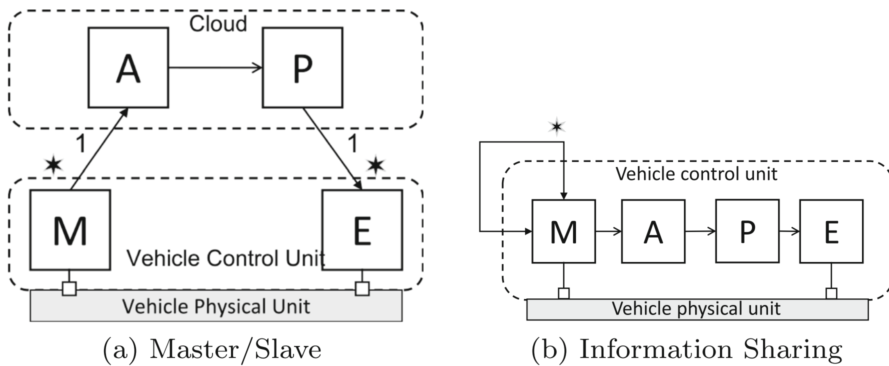


Fig. 12 Control patterns for the ITS scenario

7 DECOR in practice

In this Section, we use DECOR in three different application scenarios. The purpose is twofold: (1) to demonstrate the expressiveness of MAPE- ML, and (2) to demonstrate the ability to deal, at the Managed System level, with different application domains and their domain-specific quality attributes. In particular, the ITS (Sect. 7.1) is used to demonstrate how to evaluate two different MAPE-K patterns and observe their impact on the managed system. A Smart Power Grid (Sect. 7.2) is used to demonstrate the ability of DECOR to deal also with continuous systems. Finally, a Cloud Computing System (Sect. 7.3) is used to demonstrate how to leverage DECOR for testing and evaluating different adaption policies.

7.1 Intelligent transportation system

Scenario ITS is described in Sect. 2.

Goal The Desired Quality Attribute Measure for the ITS is formulated in Table 3. The goal is to experiment with different MAPE-K Patterns and: (1) check whether they can achieve the *Response Measure* "Ambulance travel time $<$ 50 seconds", (2) assess the MAPE-K Controls behavior in terms of adaptation time and the number of interactions among the MAPE components.

Managed system model The managed system is modeled and simulated using MovSim [51], a multi-model and open-source simulator for lane-based microscopic traffic. Since MovSim does not allow exporting models in FMU, we followed the same approach used to integrate PeerSim (see Fig. 11). In particular, we implemented a MovSim Artifact to expose a DEVS view of the MovSim mode by implementing the functions described in Sect. 6.2. Once MoveSim is integrated into MECSYCO, the ITS model can be co-simulated with the Control Model.

Control Model. The Control model can be modeled according to any Control Pattern in the patterns catalog. However, *Coordinated Pattern* and *Regional Planning Pattern* are not appropriate for the given scenario:

- *Coordinated Pattern* requires all the M, A, P, and E components to be deployed on each vehicle and to coordinate with corresponding components of other vehicles. As the number of vehicles can be very large, the total number of M, A, P, and E components (and the interactions among them) grow exponentially and become intractable.
- *Regional planning* use regional planners (i.e., the P components responsible for each region) to interact with each other and coordinate global adaptions, that is adaptations that span multiple regions. This would require extra resources at the infrastructure level for physically dividing the managed road section into several regions and monitoring the borders among them. In fact, when a vehicle crosses a border, control responsibility should be transferred from one regional planner to another without interruption of service.

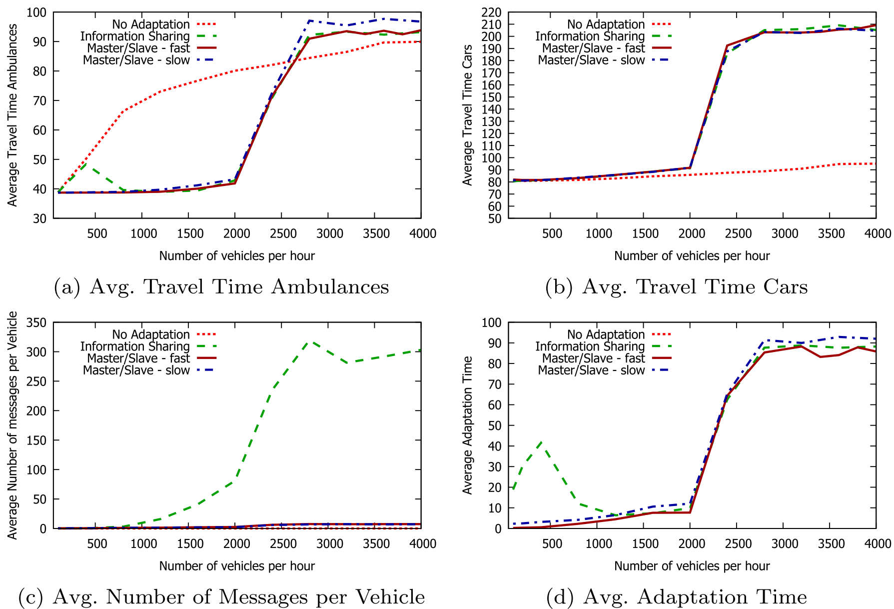
On the other hand, both *Master/Slave Pattern* and *Information-sharing Pattern* might represent a viable choice for implementing the control scenario.

Co-simulation. The Desired Quality Attribute Measure is evaluated by experimenting with the proposed MAPE-K control patterns (see Fig. 12). In the *Master/Slave* control pattern, where the cloud is responsible for ingesting the data, we tested two network configurations with different latency: (i) the first configuration models a network characterized by a latency value between 100ms and 150ms, whereas (ii) the second configuration models a network characterized by a latency value between 1s and 1.5s.

On the other hand, for the *Information Sharing* control pattern, we rely on a vehicle-to-vehicle wireless network with a communication range of 200$meters$ and latency between 3 and 10ms. These values comply with state-of-the-art vehicular wireless protocols like 802.11p [26].

In Fig. 13 we report the performance results of the two MAPE-K control patterns. Figure 13a shows that there are no substantial differences between the *Master/Slave* and *Information Sharing* control patterns regarding the average travel time of ambulances. In particular, Fig. 13a shows that for every architecture, the Desired Quality Attribute Measure is satisfied up to $\approx$ 2200 vehicles, which indicates that the identified metric is not latency-sensitive.

A noteworthy outcome of this experiment is that with over $\approx$ 2500 vehicles per hour crossing the road section, our adaptation procedure performs worse than in case of no adaptation. We analyzed this result deeper and found that the continuous trigger of the self-adaptation procedure saturates the left lane by causing a bottleneck of the two lanes road (even on the right lane). In this unfortunate case, the ambulances are trapped by the same self-adaptive system that, instead, was intended to speed up their transit. Another result of this experiment is the behavior of the *Information Sharing* pattern for traffic conditions around 500 vehicles per hour. The performance degradation peak shown in Fig. 13a and d is caused by the lack of active network links for propagating the messages. When the number of vehicles is low, the distance between two vehicles might exceed 200 meters (i.e., the communication range between vehicles). The intr-vehicle distance affects the correct propagation of the messages over the vehicle-to-vehicle network and prevents triggering the adaptation. A possible solution to this problem would be to set a retransmission timeout.



(a) Avg. Travel Time Ambulances (b) Avg. Travel Time Cars

(c) Avg. Number of Messages per Vehicle (d) Avg. Adaptation Time

Fig. 13 Intelligent transportation system - experiments

Figures 13b and c illustrate the cost of self-adaptation. In particular, Figure 13b shows how the average travel time of cars is always penalized in adaptive scenarios. Further, Figure 13c reports the cost of the average number of messages per vehicle sent for the experimented MAPE-K control patterns. The *Information Sharing* pattern is the most expensive, with up to 300 messages per vehicle sent when the road is fully saturated.

Finally, Figure 13d depicts the average adaptation time for the self-adaptive architectures, measured as the time between an ambulance sends a message until all the cars move to the left lane (i.e., from the start to the end of the adaptation). Such a metric is intrinsically measured by the average travel time (see Fig. 13a), hence the same considerations that we did before hold.

Analysis. The architect can reason on the system and obtain important insights about the experimented scenarios. In particular, for the defined, planned adaptation, the Desired Quality Attribute Measure is satisfied in a certain operational range (i.e., between 0 and 2000 vehicles per hour). Centralized planning with a slow connection to the planning component has a minimal effect on the average travel time of ambulances, and the Desired Quality Attribute Measure is still satisfied. Adaptation introduces an unavoidable overhead to the system. Specifically, the average travel time of a car always increases. The engineer is then required to consider such results in conjunction with other system requirements. Finally, the *Information Sharing* control pattern yields overhead with the high number of messages sent through the network. With such measures, we may examine if the wireless communication channel can support such a load without degradation. A more detailed network protocol model could take this aspect into account in the next design-evaluate cycle of DECOR.

Table 4 Desired quality attribute measure for the SPG scenario

Source	Power grid component
Stimulus	Ground fault detection
Artifact	The control subsystem
Environment	Runtime operating environment with any work-load
Response	Flow routed through a non faulty path
Response measure	Fault recovery time
Variants	Decentralized control
Valid configurations	Not specified
Fragment constraints	Not specified

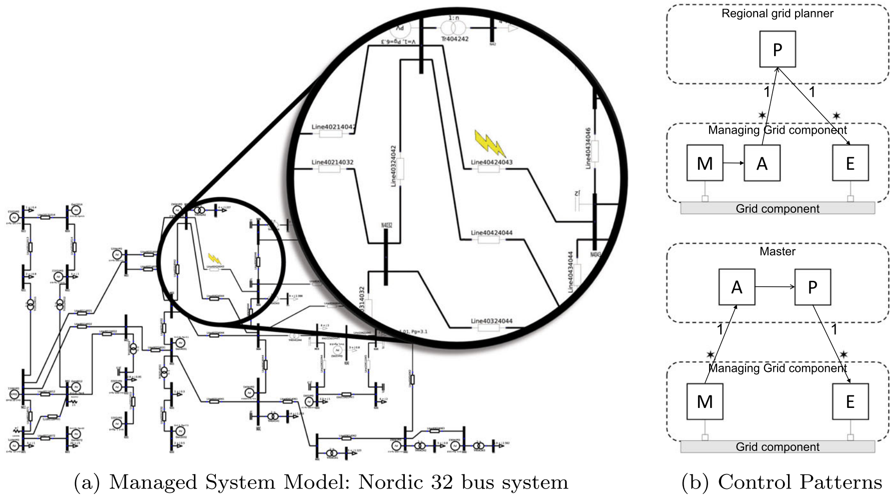
7.2 Smart power grid

Scenario A Smart Power Grid (SPG) is a cyber-physical system integrating the electrical power grid (i.e., managed system) with information and communication infrastructures for managing stable and sustainable electric energy provision (i.e., managing system). To guarantee the provision of stable and sustainable energy, the SPG should support adaptive behavior and mitigate run-time uncertainty by controlling the power distribution. A system quality requirement for the SPG is "the power grid shall be able to continue operating with minimal interruption whether components fail".

Goal From the above general requirement, we can derive several different Desired Quality Attribute Measures, one for each type of failure occurring in a power grid. Table 4 shows the Desired Quality Attribute Measure for a "Ground Fault". A ground fault is an inadvertent contact between the energized conductor and the ground, usually because of insulation breakdown due to, for instance, damaged appliances, worn wire insulation, or cable cutting. When a ground fault is detected, the affected line should be isolated, and the electricity flow should be routed through an alternative path (if existing). The goal is to (1) assess the behavior of the SPG (with and without self-adaptation) in case of a ground fault and measure the *fault recovery time* (i.e., the time needed to detect, isolate, and recover from the ground fault), and (2) evaluate the performance of different MAPE-K Controls.

Managed system model Figure 14a shows the model of the Nordic 32 bus system [50], representing an approximation of the Swedish and Nordic Power systems. The model involves 20 generators, 32 transmission, and 22 distribution buses. It includes 102 branches, among which 22 distribution and 20 step-up transformers. The Managed System is modeled in Modelica [30] and imported in DECOR as FMU.

Control model In order to achieve the stated Desired Quality Attribute Measure, the SPG shall (*a*) Monitor the actual behavior of the components in the grid (e.g., buses, generators, transmission lines), (*b*) Analyze the data, and detect the Ground Fault, (*c*) Plan the needed reconfiguration actions (i.e., switching distribution path), and (*d*) Execute the grid reconfiguration. In real-world settings, self-adaptation cannot be enacted autonomously due to safety reasons and regulations. Instead, an operator



(a) Managed System Model: Nordic 32 bus system (b) Control Patterns

Fig. 14 Smart power grid

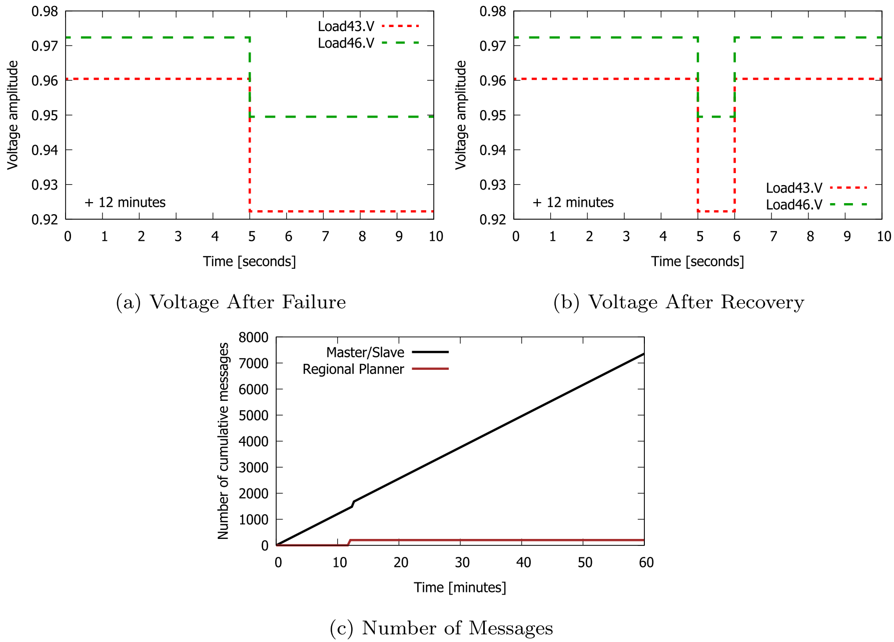
should check and validate the adaptation plan before being executed. This restricts our design space, and we consider only *Master/Slave* and *Regional Planning* patterns as both patterns have centralized planning.

Figure 14b (top side) shows the Control Model organized according to the *Regional Planning* pattern, with multiple *Managed Subsystem*, one for each component in the grid (i.e., Managing grid components), and a single *Regional Planner* component which aims at helping the operator, that is, the regional grid planner providing optional reconfigurations [32]. The managing grid components *monitor* and *analyze* local grid information (e.g., voltage amplitude, voltage angle), trigger adaptations if needed, and *execute* the adaptations. On the other hand, depicted in Fig. 14b (bottom side), in the Control Model organized by the *Master/Slave* pattern, the Managing Grid Components *monitor* the local grid components and send data to the *Master*. The *Master* analyzes the data and plans the adaptation actions when needed. The adaptation is then forwarded to and executed locally by the Managing Grid component.

Co-simulation We run two experiments: the first aims at assessing the behavior of the grid with and without self-adaptation after a ground fault, whereas the second aims at evaluating the two MAPE-K controls described above.

During the simulation, we inject a ground fault at the line between bus $N4042$ and bus $N4043$ (see the lightning symbol in zoomed area of Fig. 14a) at simulation time $t = 12m05s$. The *Managed Subsystem* components managing the bus monitor and analyze the voltage every $1s$ and, if necessary (i.e., the value deviates from a standard known threshold), communicate the monitored information to the *Regional Planner* through a network with latency in the range $30 - 80ms$ and no packet loss.

In the first experiment, we measured the ground fault's effect by observing the voltage amplitude level of $Load43$ and $Load46$. Figure 15a shows the behavior of the two components in normal operation, i.e., without self-adaptation. In particular, after



(a) Voltage After Failure

(b) Voltage After Recovery

(c) Number of Messages

Fig. 15 Smart power grid - experiments

the failure happens at $t = 12m05s$, the voltage of the loads deviates from its expected p.u. value[5].

We then assess the self-adaptive capability of the SPG. The Control is aware of the altered voltage of the loads' value. It knows which component is causing the fault and devises a reconfiguration plan using this information. The plan consists in reconfiguring the energy distribution path to isolate the faulted line. Figure 15b shows how, thanks to the reconfiguration, the voltage level at the loads is re-established quickly. Note that with both the *Master/Slave* and the *Regional Planning*, we observed the same results (showed in Fig. 15a and b. This observation is because the Managing Grid component monitors the value of the voltage every $1s$. Then, in both cases, the fault is detected at simulation time $t = 12m06s$, and the *fault recovery time* is less than 1 second.

The second experiment evaluates the advantages/disadvantages of adopting the two control patterns. To this end, we analyze the number of cumulative messages sent through the network by the group of components realizing the self-adaptive behavior via the two control strategies (see Fig. 15c).

Analysis Figure 15c shows that even though both patterns satisfy the stated Desired Quality Attribute Measure, *Regional Planning* pattern outperforms the *Master/Slave* one. In fact, since *Regional Planning* analyses the data at the edge of the network (within the *Managing Grid components*), messages that are not relevant for the adap-

[5] In the power systems analysis field of electrical engineering, the numerical per-unit value of any quantity is its ratio to a chosen base quantity of the same dimension [28].

Table 5 Desired Quality Attribute Measure for the CC scenario

Source	Power grid component
Stimulus	Load greater than a given threshold q
Artifact	The control subsystem
Environment	Runtime operating environment with any work-load
Response	Spawn new virtual machines
Response Measure	Response time $< 10s$
Variants	Decentralized control
Valid configurations	Not specified
Fragment constraints	Not specified

tation are managed locally. Whereas, in the *Master/Slave* pattern, the number of cumulative messages sent through the network linearly increases. In real cases, where the number of managed components in the power grid is very large, this could lead to scalability issues.

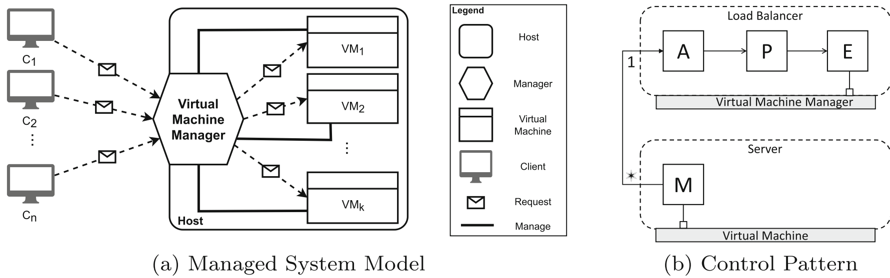
7.3 Cloud computing system

Scenario Cloud Computing (CC) concerns the on-demand availability of computer system resources without direct active management from the user. In this application scenario, we experiment with a system simulating the characteristic of the SEAMS exemplar "`zzn.com`" [19]. The `zzn.com` website uses a load balancer to balance requests across a pool of replicated servers, the size of which is dynamically adjusted to balance server utilization against service response time. A quality requirement for the CC scenario can be "When the load on the system increases, a new virtual machine shall be spawned so that the response time for requests is always lower than 10 seconds".

Goal From the above quality requirement, we can derive the Desired Quality Attribute Measure specified in Table 5. The goal is to experiment with different adaptation policies and understand how the system behaves when varying some configuration parameters.

Managed system model Figure 16a shows CC scenario described above modeled in CloudSim [13], an extensible simulation toolkit for modeling and simulating cloud infrastructures and services. Specifically, clients c_j send requests to the Virtual Machine Manager (VMM) hosted in the cloud, which in turn balances the load by forwarding requests to Virtual Machines (VM_i) in a service tier. The VMM is also in charge of managing the VM_i lifecycle (i.e., create, modify, monitor, start, and stop). CloudSim does not export models in FMU. Therefore, as done for the ITS, we implemented a CloudSim Artifact and integrated it into MECSYCO.

Control model To satisfy the Desired Quality Attribute Measure, the proposed control pattern shall continuously monitor the performance of each VM_i, detect the congestion of resources and reconfigure the pool of available virtual machines. To this end, Figure 16b shows a custom control pattern where each VM_i is in charge of



(a) Managed System Model (b) Control Pattern

Fig. 16 Cloud computing system

monitoring the response time for each served request and sending such a measure to the VMM, which in turn analyzes the data and, if necessary, plans and executes a reconfiguration for the pool of managed VM_i.

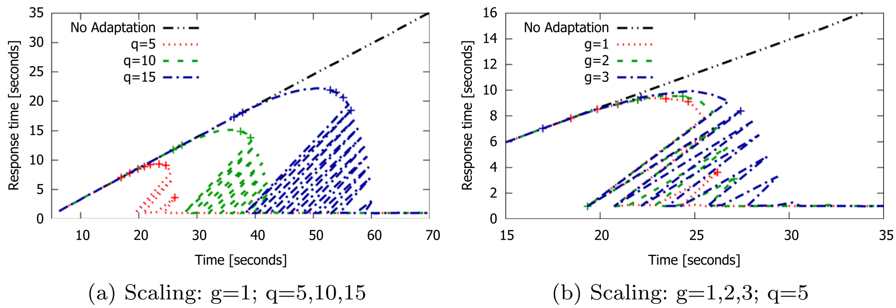
Co-simulation In our experimentation, the pool of VM_i is initialized with a single VM able to process multiple requests according to the time-shared scheduling policy. We adopt the simplifying assumption of uniform requests (i.e., requests of the same size). There exists only one type of VM which can respond to a request in isolation in $1sec$. The VMM load balances the requests to the pool of VM_i by adopting the round-robin policy. A new request is submitted to the system every $700ms$ from $t = 5s$.

Let r be a request submitted to the VMM and $RT_i(r)$ be the response time of the virtual machine VM_i serving the request r. Every time a request r is successfully processed by a virtual machine VM_i, the resulting response time $RT_i(r)$ is communicated to the VMM (see the arrow from M to A in Fig. 16b).

The analyzing component of the VMM aggregates the last w response times observed from the VM_i pool in D_w. Similar to [21], the concept of learning window w avoids giving too much weight to old historical data, which would be inappropriate in a highly dynamic environment. The P component of the VMM predicts the expected response time of a VM in the pool of resources by calculating the average value of the last w observations, $L = D_w/w$.

Finally, the designed scaling policy works as follows: let q be a threshold quality value for the response time, S_t the time of the last scaling operation and, g be the grace period between two scaling operations (i.e., the minimum time that should elapse between scaling commands). At the time t, a new VM is launched if $L > q$ and $t > S_t + g$.

In the first experiment, see Fig. 17a, we experiment with the proposed control pattern by setting $g = 1$ and test the proposed scaling policy for different threshold quality values $q = 5, 10, 15$. The baseline curve shows how the response time of the single virtual machine linearly increases if a scaling policy is not adopted (i.e., the system does not self-adapt to the emerging conditions). The response time of the other curves, representing the q-parameterized scaling policies, is symmetric. Note that by increasing the threshold value q, the response time for the pool of resources increases. We can see that only the scaling policy with $q = 5$ is able to satisfy the Desired Quality Attribute Measure (i.e., the response time of requests is lower than $10s$).



(a) Scaling: g=1; q=5,10,15

(b) Scaling: g=1,2,3; q=5

Fig. 17 Cloud Computing - Experiments

An exciting outcome of the experiment appears by looking at the number of scaling operations executed by the policies, reported with cross marks on the respective curves. In particular, we can see that for $q = 5$, the system can keep control of the incoming load by satisfying the Desired Quality Attribute Measure. However, it does this by increasing the pool of resources by 8 virtual machines between $15 - 20s$. For $q = 10$ and $q = 15$, the number of scaling operations is 4 and 6, respectively.

We then experiment with $q = 5$ and see if such a policy can achieve the same performance (i.e., still satisfy the Desired Quality Attribute Measure) by issuing fewer VMs. Frequent scale-up operations might incur more costs and cause more energy consumption for the CC infrastructure. To this end, we experiment with different values of grace periods $g = 1, 2, 3$. Referring to Fig. 17b we can see that the system can still satisfy the Desired Quality Attribute Measure for all the experimented cases. Moreover, for $g = 3$ the policy instantiates only 3 VMs against the 8 VMs adopted by the policy with $g = 1$ and 5 VMs for the policy with $g = 2$.

Analysis The main insight obtained from the simulation results in Fig. 17 is that the defined scaling policy imposes a trade-off between the quality attribute (i.e., response time) and the overhead cost (i.e., number of launched VMs). The operational point of the adopted adaptation policy must then be selected concerning the Desired Quality Attribute Measure at hand. In this scenario, we did not experiment with different MAPE-K control patterns. The identification of the responsibility structures for the managing components entails the adoption of one specific control. However, we have shown how DECOR allows not only for evaluating and choosing the desired MAPE-K control but also for improving the designed adaptation policy once a particular architecture is instantiated. In particular, in this scenario, the engineer selected the control structure and explored how the adaptation policy behaves initially by varying a single parameter (see Fig. 17b). In the second round, the behavior of the policy is further explored to reduce its overhead while still respecting the Desired Quality Attribute Measure (see Fig. 17a).

8 Discussion

Objectives **O1** and **O2** have been achieved by fulfilling the set of elicited requirements (see Sect. 1): **R1.1** *Multi-paradigm Modeling*, **R1.2** *Decentralized Control* modeling, and **R2** *Evaluation* of control alternatives.

The insights obtained in each of the experimented scenarios (see Sect. 7) demonstrate how DECOR can be used to reason about the self-adaptive system under investigation and to determine whether the given control achieves critical quality attribute requirements. Indeed, for each experimented system, (i) we investigated whether the control and/or the adaptation policy under examination can satisfy the Desired Quality Attribute Measure, and (ii) we analyzed the cost of the considered control in terms of different quality measures (e.g., adaptation time, number of exchanged messages).

The multi-paradigm modeling support provided by DECOR (i.e., **R1.1**) allows for reasoning about different types of systems: it is possible to model closed systems, where the model is known at design time and does not change over time (e.g., as in the SPG scenario), as well as open systems, where the model is inherently dynamic and continuously changes its structure and behavior (e.g., in the ITS scenario). It is possible to model cyber-physical systems (e.g., ITS and SPG scenarios) and ICT systems (e.g., CC scenario).

The DECOR CME provides a graphical concrete syntax for modeling decentralized control (i.e., **R1.2**). Specifically, we modeled *Master/Slave* and *Information Sharing* patterns in the ITS scenario, *Master/Slave* and *Regional Planning* patterns in the SPG scenario, and a *Custom Control*, which does not adhere to any pre-specified pattern, in the CC scenario.

Finally, we showed how to evaluate the designed multi-paradigm models through co-simulation (i.e., **R2**). The simulation results considered quality attributes measurements specific to the employed control (e.g., average adaptation time), as well as quality attributes measurements that depend on the specific type of managed system examined (e.g., voltage amplitude). In each case, we have shown how different architectures and self-adaptation policies can lead to different performance measures. For the sake of simplicity and understandability, in all experimented scenarios, the DECOR has been used to evaluate only a single MAPE-K control. However, it is worth noticing that DECOR also allows for modeling multiple concurring MAPE-K patterns and observing through simulation whether they affect each other. For example, in case of conflicting MAPE-K, the simulation results would show the system not behaving as expected and not fulfilling the stated Desired Quality Attribute Measure. A thorough investigation of these aspects is left for future work.

8.1 Threats to validity

There are some potential threats to the validity of the proposed approach.

A threat to internal validity might be that we defined a single Desired Quality Attribute Measure for each of the addressed application scenarios. A given control may satisfy a Desired Quality Attribute Measure but invalidate or affect the realization of other Desired Quality Attribute Measures. To this end, it is worth noticing that DECOR allows for adding a new Desired Quality Attribute Measures at each design-evaluate cycle and re-evaluating the control according to the new set of requirements. However, analyzing each requirement in isolation might lead to inconsistencies. A control mechanism designed for addressing a specific quality aspect might not be

optimal for a recently introduced Desired Quality Attribute Measure. On the other side, evaluating all the quality requirements in a single design-evaluate cycle can make the reasoning very complex. One strategy to tackle this problem might be: (i) start with all the Desired Quality Attribute Measures, (ii) if the problem becomes intractable, group similar Desired Quality Attribute Measures and execute a design-evaluate cycle for addressing them, (iii) evaluate the identified control with respect to the remaining Desired Quality Attribute Measures. This is only an example strategy, and other divide-et-impera approaches might be as effective. Hence, to mitigate the threat to internal validity, a future investigation might be based on finding and documenting state-of-the-art approaches. Further, it is worth mentioning that, if it is not possible to satisfy all the identified Desired Quality Attribute Measures, a prioritization of requirements based on user-defined preferences is required.

Another threat to internal validity is the integration of third-party simulators in the DECOR CoSE. One of the main design decisions we made for implementing CoSE, is to rely on FMU/FMI [8], as it is a well-known and broadly used standard for specifying and exchanging simulation models[6]. However, only one (i.e. SPG) of the three scenarios discussed in Sect. 7 exploits such a functionality. For the other scenarios – i.e., ITS, and CC – we had to re-engineer the used simulators – MovSim and CloudSim, respectively – and manually integrate them into MECSYCO. In general, if the given simulator leverages discrete event simulation, the integration procedure is easy and does not require much effort. However, it is not as straightforward as with FMU/FMI and requires (i) to have access to the simulator source code and (ii) to re-design the event processing routine so that the simulation can be progressed step-wise by MECSYCO[7].

A threat to external validity concerns the approach evaluation. Indeed, we demonstrated the framework's applicability by using it in three application scenarios. Although the three scenarios are different in nature and have different requirements and characteristics, this is not enough to validate the generality of the approach. Indeed, a more extensive and deep evaluation is required. To mitigate such a threat, we plan, as part of future work, to conduct a set of controlled experiments. In particular, we plan to identify a few real use cases considering multiple domains, requirements, and design dimensions. Then, a group of professionals and students will be selected from ICT companies in the Växjö area and from the *Software Technology Master Programme* at *Linnaeus University*, respectively. Each identified use case will be addressed by 2 teams of students (T_1^S, T_2^S) and 2 teams of professionals (T_1^P, T_2^P), by using different approaches: T_1^S and T_1^P will make use of DECOR, T_2^S and T_2^P will make use of approaches/tools from the state-of-the-art (see Sect. 3). During the development, we will constantly monitor the different teams and gather some data. The objective of the study is manyfold and aims at: (i) evaluating the generality and applicability, as well as understanding the limitations of the proposed approach, (ii) assessing the number of design-evaluate cycles needed to satisfy all the quality requirements when using DECOR and the other approaches/tools, (iii) measuring the time spent for modeling the MAPE-K Control in DECOR and other tools, (iv) assessing the usability of the DECOR

[6] https://fmi-standard.org

[7] For more details, see https://github.com/mi-da/DECOR

tools (e.g., MAPE- ML/CME, and CoSE). Indeed, all these aspects will be addressed from the professionals' and students' points of view.

9 Conclusions and future work

Designing decentralized control architectures is challenging: architects should consider many different and interdependent design dimensions to devise a control able to timely perform the correct adaptations and make the system meet its goals. To this end, several different alternatives might exist to architect the control. However, different control alternatives entail different properties (e.g., performance and cost), which should be carefully evaluated to select the best candidate for the given system.

In this paper, we presented DECOR, a model-based reasoning framework for designing and evaluating decentralized control. In particular, DECOR provides (i) multi-paradigm modeling support, (ii) a modeling environment for architecting MAPE-K decentralized control, and (iii) a co-simulation environment for simulating the decentralized control together with the managed system and estimating the quality attributes of interest. Therefore, DECOR allows architects to model the decentralized control along with the managed system and to evaluate how they affect each other.

We used DECOR to model and evaluate three different application scenarios: an intelligent transportation system, a smart power grid, and a cloud computing system. In these scenarios, we demonstrated how DECOR allows architects to make informed design decisions on the decentralized control and adaptation strategies to satisfy the system requirements.

Ongoing and future work proceeds toward different lines of research. First, we aim to consolidate both the CME and CoSE tools by, e.g., removing bugs, improving the GUI, and providing an installation package. Further, though intuitive, the Ecore representation of the MAPE- ML Metamodel does not give a precise semantic description of the language constructs, which is instead available in FORMS. First, we plan to provide MAPE- ML with denotational semantics by defining a mapping from MAPE- ML to Z [20], the formal language used for specifying FORMS. Second, we aim to provide developers with the ability to specify the behavior of the managing components formally and automatically deploy them (e.g., [35]). In this way, developers could define the algorithms determining the concrete behavior of the MAPE components at design time. Further, as in [53], we envision the development/adoption of a validation technique able to test and analyze the control behavior at run time and automatically detect unintended interactions. We also aim to investigate how to evolve/adapt the defined control itself. The run-time adaptation of the collective control might be necessary due to changes in the system resources, the environment, or the system goals. Finally, as remarked in Sect. 8.1, we plan to conduct a set of controlled experiments with professionals and students to conduct an extensive and deep evaluation of the approach's generality.

Funding Open access funding provided by Linnaeus University.

Competing Interests The authors have no relevant financial or non-financial interests to disclose.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Abbas N, Andersson J (2013) Architectural reasoning for dynamic software product lines. In: Proceedings of the 17th International Software Product Line Conference Co-Located Workshops
2. Abbas N, Andersson J, Weyns D (2020) Asple: A methodology to develop self-adaptive software systems with systematic reuse. J Syst Softw 167
3. 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)
4. Bass L, Clements P, Kazman R (2003) Software Architecture in Practice. Addison-Wesley
5. Bass L, Clements P, Kazman R (2012) Software Architecture in Practice, 3rd edn. Addison-Wesley Professional, Boston
6. Bass L, Ivers J, Klein M, Merson P (2005) Reasoning frameworks. Tech. Rep. CMU/SEI-2005-TR-007, Software Engineering Institute, Carnegie Mellon University
7. Bernardo M, Hillston J (2007) (eds.): Formal Methods for Performance Evaluation, vol. 4486
8. Blochwitz T, et al (2012) Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models. In: Proceedings of the 9th International Modelica Conference. The Modelica Association
9. Bolchini C et al (2013) A framework to model self-adaptive computing systems. In: Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems
10. Boronat A, Knapp A, Meseguer J, Wirsing M (2008) What is a multi-modeling language? In: International Workshop on Algebraic Development Techniques. Springer
11. Brambilla M, Cabot J, Wimmer M (2017) Model-driven software engineering in Practice, 2nd edn. Morgan & Claypool Publishers
12. Brun Y, Desmarais R, Geihs K, Litoiu M, Lopes A, Shaw M, Smit M (2013) A Design Space for Self-Adaptive Systems, pp. 33–50. Springer Berlin Heidelberg, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-35813-5_2
13. Calheiros RN, Ranjan R, Beloglazov A, De Rose CAF, Buyya R (2011) Cloudsim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. Softw. Pract. Exper. 41(1)
14. Calinescu R, Gerasimou S, Banks A (2015) Self-adaptive software with decentralised control loops. In: Egyed A, Schaefer I (eds.) Fundamental Approaches to Software Engineering
15. Camus B, Bourjot C, Chevrier V (2015) Combining devs with multi-agent concepts to design and simulate multi-models of complex systems (wip). In: Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium, DEVS '15, pp. 85–90. Society for Computer Simulation International, San Diego, CA, USA
16. Camus B, Vaubourg J, Presse Y, Elvinger V, Paris T, Tan A, Chevrier V, Ciarletta L, Bourjot C Multi-agent environment for complex systems cosimulation - architecture documentation. http://mecsyco.com/dev/doc/
17. Camus B et al (2016) Hybrid co-simulation of FMUs using DEV&DESS in MECSYCO. In: Proceedings of the Symposium on Theory of Modeling & Simulation, TMS/DEVS 2016, Pasadena, CA, USA, April 3-6, 2016
18. Cheng BHC, 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 HM, Kramer J, Litoiu M, Malek S, Mirandola R, Müller HA, Park S, Shaw M, Tichy M, Tivoli M, Weyns D, Whittle J (2009) Software Engineering for Self-Adaptive Systems: A Research Roadmap, pp. 1–26. Springer Berlin Heidelberg, Berlin, Heidelberg

19. Cheng S, Garlan D, Schmerl B (2009) Evaluating the effectiveness of the rainbow self-adaptive system. In: 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems
20. Community Z Tools: CZT. http://czt.sourceforge.net/ (2016)
21. D'Angelo M, Caporuscio M, Grassi V, Mirandola R (2020) Decentralized learning for self-adaptive qos-aware service assembly. Future Generation Computer Systems 108
22. D'Angelo M, Caporuscio M, Napolitano A (2017) Model-driven engineering of decentralized control in cyber-physical systems. In: Proceedings of the 2nd IEEE International Workshops on Foundations and Applications of Self* Systems (FAS*W)
23. D'Angelo M, Gerasimou S, Ghahremani S, Grohmann J, Nunes I, Pournaras E, Tomforde S (2019) On learning in collective self-adaptive systems: State of practice and a 3d framework. In: 2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)
24. D'Angelo M, Napolitano A, Caporuscio M (2018) Cyphef: A model-driven engineering framework for self-adaptive cyber-physical systems. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings
25. De Lemos R et al (2013) Software engineering for self-adaptive systems: A second research roadmap. In: Software Engineering for Self-Adaptive Systems II
26. Demmel S, Lambert A, Gruyer D, Rakotonirainy A, Monacelli E (2012) Empirical ieee 802.11 p performance evaluation on test tracks. In: Proceedings of International Conference on Intelligent vehicles symposium (IV)
27. Diaz-Pace A, Kim H, Bass L, Bianco P, Bachmann F (2008) Integrating quality-attribute reasoning frameworks in the arche design assistant. In: Becker S, Plasil F, Reussner R (eds.) Quality of Software Architectures. Models and Architectures, pp. 171–188. Springer Berlin Heidelberg, Berlin, Heidelberg
28. El-Hawary ME (2008) Introduction to Electrical Power Systems. Wiley-IEEE Press
29. Elkhodary A, Esfahani N, Malek S (2010) Fusion: A framework for engineering self-tuning self-adaptive software systems. In: Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE '10. Association for Computing Machinery
30. Fritzson P, Bunus P (2002) Modelica – A General Object-Oriented Language for Continuous and Discrete-Event System Modeling. In: Proceedings of the 35Th Annual Simulation Symposium
31. Garlan D et al (2004) Rainbow: architecture-based self-adaptation with reusable infrastructure. Computer 37(10)
32. Haghighat H, Zeng B (2016) Distribution system reconfiguration under uncertain load and renewable generation. IEEE Transactions on Power Systems 31(4)
33. Hardebolle C, Boulanger F (2009) Exploring multi-paradigm modeling techniques. SIMULATION 85(11–12):688–708
34. Hezavehi SM, Weyns D, Avgeriou P, Calinescu R, Mirandola R, Perez-Palacin D (2021) Uncertainty in self-adaptive systems: A research community perspective. ACM Trans. Auton. Adapt. Syst. 15(4)
35. Iftikhar MU, Weyns D (2014) ActivFORMS: Active formal models for self-adaptation. In: Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems
36. Im T, Vullam S, McGregor JD (2010) Reasoning about safety during software architecture design. In: Rahal I, Zalila-Wenkstern R (eds.) ISCA 19th International Conference on Software Engineeringand Data Engineering (SEDE-2010) June 16-18, 2010, Hilton Fisherman's Wharf, San Francisco, CA, USA, pp. 1–8. ISCA
37. Kephart JO, Chess DM (2003) The vision of autonomic computing. Computer 36(1)
38. Kit M, Gerostathopoulos I, Bures T, Hnetynka P, Plasil F (2015) An architecture framework for experimentations with self-adaptive cyber-physical systems. In: 2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems
39. Krupitzer C, Roth FM, Becker C, Weckesser M, Lochau M, Schürr A (2016) Fesas ide: An integrated development environment for autonomic computing. In: 2016 IEEE International Conference on Autonomic Computing (ICAC)
40. Malek S, Edwards G, Brun Y, Tajalli H, Garcia J, Krka I, Medvidovic N, Mikic-Rakic M, Sukhatme GS (2010) An architecture-driven software mobility framework. J Syst Softw 83(6):972–989
41. Micouin P (2014) Model-Based Systems Engineering: Fundamentals and Methods. Wiley
42. Montresor A, Jelasity M (2009) PeerSim: A scalable P2P simulator. In: Proc of the 9th Int Conference on Peer-to-Peer (P2P'09)

43. Moreno GA, Merson P (2008) Model-driven performance analysis. In: Becker S, Plasil F, Reussner R (eds.) Quality of Software Architectures. Models and Architectures, pp. 135–151. Springer Berlin Heidelberg, Berlin, Heidelberg
44. Raibulet C, Arcelli Fontana F, Capilla R, Carrillo C (2017) Chapter 13 - an overview on quality evaluation of self-adaptive systems. In: Mistrik I, Ali N, Kazman R, Grundy J, Schmerl B (eds.) Managing Trade-Offs in Adaptable Software Architectures, pp. 325–352. Morgan Kaufmann, Boston. https://doi.org/10.1016/B978-0-12-802855-1.00013-7
45. Salehie M, Tahvildari L (2009) Self-adaptive software: Landscape and research challenges. ACM Trans. Auton. Adapt. Syst. 4(2)
46. Schmidt DC (2006) Model-driven engineering. IEEE. Computer 39(2)
47. Siebert J, Ciarletta L, Chevrier V (2010) Agents and artefacts for multiple models co-evolution: Building complex system simulation as a set of interacting models. In: Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: Vol. 1 - Volume 1, pp. 509–516. International Foundation for Autonomous Agents and Multiagent Systems
48. Sommer C et al (2008) On the need for bidirectional coupling of road traffic microsimulation and network simulation. In: Proceedings of the 1st ACM SIGMOBILE workshop on Mobility models
49. Steinberg D, Budinsky F, Paternostro M, Merks E (2009) EMF: Eclipse Modeling Framework 2.0, 2nd edn. Addison-Wesley Professional, Boston
50. Stubbe M Long-term dynamics - phase II. Report of CIGRE Task Force 38.02.08
51. Treiber M, Kesting A (2010) An open-source microscopic traffic simulator. IEEE Intelligent Transportation Systems Magazine 2(3)
52. Treiber M, Kesting A, Helbing D (2010) Three-phase traffic theory and two-phase models with a fundamental diagram in the light of empirical stylized facts. Trans Res Part B: Methodol 44(8):983–1000
53. Viroli M, Bucchiarone A, Pianini D, Beal J (2016) Combining self-organisation and autonomic computing in cass with aggregate-mape. In: 2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W)
54. Vogel T, Giese H (2014) Model-driven engineering of self-adaptive software with eurema. ACM Trans. Auton. Adapt. Syst. 8(4)
55. Von Meier A (2006) Electric Power Systems: A Conceptual Introduction. Wiley Survival Guides in Engineering and Science, Wiley
56. Warmer J, Kleppe A (2003) The Object Constraint Language: Getting Your Models Ready for MDA, 2nd edn. Addison-Wesley Longman Publishing Co., Inc, USA
57. Wätzoldt S, Giese H (2015) Modeling collaborations in adaptive systems of systems. In: Proceedings of the 2015 European Conference on Software Architecture Workshops, ECSAW '15
58. Weyns D, Iftikhar MU, de la Iglesia DG, 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, C3S2E '12
59. Weyns D, Malek S, Andersson J (2012) Forms: Unifying Reference Model for Formal Specification of Distributed Self-Adaptive Systems. ACM Transactions on Autonomous and Adaptive Systems 7(1)
60. Weyns D, Schmerl B, Grassi V, Malek S, Mirandola R, Prehofer C, Wuttke J, Andersson J, Giese H, Göschka KM (2013) On Patterns for Decentralized Control in Self-Adaptive Systems, pp. 76–107. Springer Berlin Heidelberg, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-35813-5_4
61. Wirfs-Brock R, McKean A (2003) Object design: roles, responsibilities, and collaborations. Addison-Wesley Professional
62. Zeigler BP, Kim TG, Praehofer H (2000) Theory of Modeling and Simulation, 2nd edn. Academic Press, Inc

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.