The Journal of Systems and Software 148 (2019) 37–55

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Tuning self-adaptation in cyber-physical systems through architectural homeostasis

Ilias Gerostathopoulos[a,*], Dominik Skoda[b], Frantisek Plasil[b], Tomas Bures[b], Alessia Knauss[c]

[a] Fakultät für Informatik, Technische Universität München, Munich, Germany
[b] Charles University in Prague, Faculty of Mathematics and Physics, Prague, Czech Republic
[c] Department of Computer Science and Engineering, Chalmers University of Technology, Gothenburg, Sweden

ARTICLE INFO

Article history:
Received 15 April 2017
Revised 2 September 2018
Accepted 29 October 2018
Available online 30 October 2018

Keywords:
Cyber-physical systems
Software architecture
Run-time uncertainty
Self-adaptation strategies
Architecture homeostasis

ABSTRACT

Self-adaptive software-intensive cyber-physical systems (sasiCPS) encounter a high level of run-time uncertainty. State-of-the-art architecture-based self-adaptation approaches assume designing against a fixed set of situations that warrant self-adaptation. As a result, failures may appear when sasiCPS operate in environment conditions they are not specifically designed for. In response, we propose to increase the homeostasis of sasiCPS, i.e., the capacity to maintain an operational state despite run-time uncertainty, by introducing run-time changes to the architecture-based self-adaptation strategies according to environment stimuli. In addition to articulating the main idea of architectural homeostasis, we introduce four mechanisms that reify the idea: (i) collaborative sensing, (ii) faulty component isolation from adaptation, (iii) enhancing mode switching, and (iv) adjusting guards in mode switching. Moreover, our experimental evaluation of the four mechanisms in two different case studies confirms that allowing a complex system to change its self-adaptation strategies helps the system recover from run-time errors and abnormalities and keep it in an operational state.

1. Introduction

Cyber-physical systems (CPS) (Kim and Kumar, 2012) are large complex systems that increasingly rely on software for their operation—they are becoming *software-intensive* CPS (Hoelzl et al., 2008; Beetz and Böhm, 2012). Such systems, e.g., in the area of smart grids, road-side computing, and intelligent transportation are typically comprised of several million lines of code. A high-level view achieved via focusing on *software architecture* abstractions is thus becoming increasingly important for dealing with such scale and complexity during development, deployment, and maintenance.

These systems continuously sense physical magnitudes in order to actuate physical processes. Due to the close connection to the physical world, whose behavior is hard to fully predict at design time and control at run-time, they encounter a high level of uncertainty in their operating conditions, called *run-time uncertainty* (Ramirez et al., 2012a). This is typically rooted in (i) unexpected changes in the run-time infrastructure (e.g., communication latencies, disconnections, sensor malfunctioning), (ii) unexpected changes in the environment (e.g., harsh weather conditions), (iii) the evolution of other cyber or physical systems that interface with the CPS in question, and (iv) the randomness introduced by human interaction. Run-time uncertainty can cause numerous failures ranging from temporary service unavailability to a complete system crash (Ramirez et al., 2012a).

A promising way to tackle run-time uncertainty is to endow software-intensive CPS with self-adaptive capabilities, i.e., with capabilities to adjust their own structure and behavior at run-time based on their internal state and the perceived environment state, while considering their run-time goals and requirements (Cheng et al., 2009). In this paper, we focus on self-adaptation at the architectural level (being inspired by (Cheng et al., 2012; Iftikhar and Weyns, 2014; Weyns et al., 2010; Floch et al., 2006; Brun et al., 2009)).

Problem statement. One of the limitations in the state-of-the-art architecture-based self-adaptation approaches is that they assume designing against a fixed set of situations that warrant self-adaptation (Gerostathopoulos et al., 2017). However, when run-time uncertainty is high, anticipating all potential situations upfront (i.e. at design time) and designing corresponding actions is a costly, lengthy, and sometimes not even a viable option (Ramirez et al., 2012a; Cheng et al., 2009).

* Corresponding author.
E-mail addresses: gerostat@in.tum.de (I. Gerostathopoulos), skoda@d3s.mff.cuni.cz (D. Skoda), plasil@d3s.mff.cuni.cz (F. Plasil), bures@d3s.mff.cuni.cz (T. Bures), alessia.knauss@chalmers.se (A. Knauss).

https://doi.org/10.1016/j.jss.2018.10.051
0164-1212/

Goal and main idea of contribution. In our work, instead of trying to identify all potential situations and corresponding actions (*strategies* in architecture-based self-adaptation), we propose to engineer flexibility in the strategies of a self-adaptive software-intensive CPS (*sasiCPS* further on) in the form of run-time changes to these strategies. This way we try to increase the software homeostasis of sasiCPS, i.e. the capacity for the system to maintain its normal operating state and implicitly repair abnormalities or deviations from expected behavior (Shaw, 2002), by specifically focusing on the architectural level—*architectural homeostasis*.

We claim that supporting architectural homeostasis at run-time helps tackle the run-time uncertainty in sasiCPS. The underlying assumptions of our approach are that (i) fixed architecture-based self-adaptation strategies result in brittle systems in domains with high run-time uncertainty, (ii) allowing the components of a complex system to change their self-adaptation strategies in a slightly different way while still aiming at a common goal, can have positive results in the overall utility of a self-adaptive system. Indeed, the last point is common in other domains (e.g., communication protocols that try to re-establish a connection in some random manner to avoid a flood of reconnections).

The main contribution of this paper[1] are four concrete homeostatic mechanisms (*H-mechanisms*) that operate at the architectural level and effectively increase the capacity of a sasiCPS to maintain an operational state despite run-time uncertainty. The secondary contribution lies in implementing the proposed H-mechanisms in a development and run-time framework for sasiCPS—*DEECo* component framework (Bures et al., 2013, 2016)—and in evaluating their feasibility and effectiveness via controlled experiments in two different case studies. In particular, we focus on answering the following research questions: (a) to which extent each H-mechanism repairs the problem it is intended to repair, (b) what is the overhead of using each H-mechanism, and (c) whether multiple H-mechanisms can be applied at the same time. The results show that using the proposed mechanisms increases the overall utility of the system in face of run-time errors and abnormalities (exceptional situations).

The rest of the paper is structured as follows: Section 2 depicts our running example and the background of our work. Section 3 presents the main idea of architectural homeostasis, together with its reification into four concrete homeostatic mechanisms. Section 4 details our evaluation based on implementing the mechanisms and quantifying their effects in the running example and in an additional case study. Section 5 elaborates on interesting points, answers our research questions and discusses the limitations of our approach. Finally, Section 6 compares our work to existing ones in the literature and the concluding Section 7 summarizes the contribution.

2. Running example and background

2.1. Cleaning robots example

In the running example used throughout the paper, four Turtlebots (http://www.turtlebot.com/) are deployed in a large 2D space with the task to keep it as clean as possible. The space is covered by tiles that can get dirty at some arbitrary points in time. Each robot is able to move around, identify dirty tiles via its downwards-looking camera and humidity sensor, and clean them. Each robot also works on a specific energy budget; before it expires, the robot needs to reach a docking station and recharge. Several docking stations exist in the space. Fig. 1 depicts a scenario with four robots and two docking stations.

[1] This paper is an extension of (Gerostathopoulos et al., 2016a).

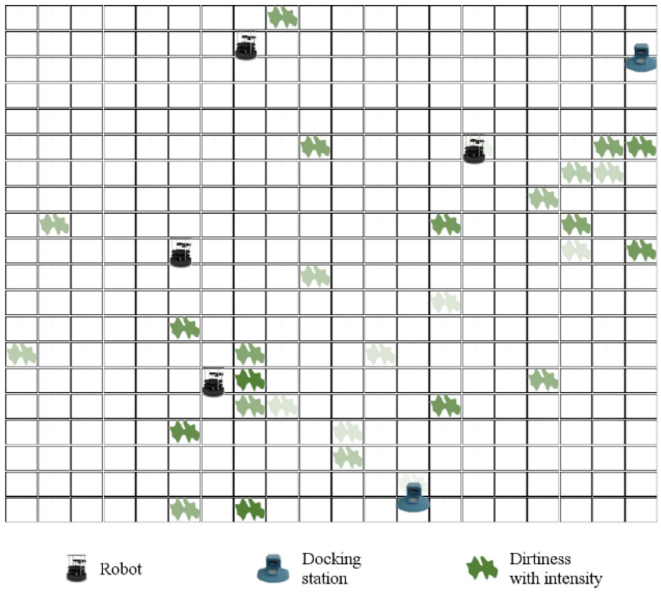


Fig. 1. Cleaning robots example (screenshot from the tool).

The robots communicate with each other to exchange information about the lastly cleaned tiles to avoid unnecessary trips. They also communicate with the docking stations to determine the most convenient station for recharging.

This example, although a toy one, comprises situations with run-time uncertainty. It includes situations where a robot loses the ability of reliably detecting dirty tiles (e.g. due to a failure in its humidity sensor) or a docking station may stop working. Run-time uncertainty is also manifested in the unpredictable rate and position where dirt appears in the space. The rate of the appearance of dirt is assumed to be an order of a magnitude smaller than the rate of information exchange between robots about the lastly cleaned tiles; thus, the exchanged information does not become stale too often.

2.2. DEECo model of cleaning robots – running example

DEECo is a development and run-time framework for sasiCPS (Bures et al., 2013). In DEECo, a component is an independent entity of development and deployment. Every DEECo component contains data (knowledge) and functionality in the form of periodically invoked processes which map input knowledge to output knowledge; each process is associated with one or more component mode(s). Two component types were identified in our running example: Robot and DockingStation. Each Robot (instance) comprises knowledge about its position, dirtinessMap, etc. (Fig. 2, lines 9–14), and several processes, e.g. clean (lines 25–31), move, and charge. A process belongs to one or more modes—e.g., the Robot's clean process belongs to cleaning mode (line 25). The modes of each component are switched at run-time according to the component's mode-state machine (such as the one in Fig. 8). A component in DEECo has a number of roles, each allowing a subset of knowledge fields to become subject to component interaction. In the running example, each Robot features the Dockable and Cleaner roles (lines 1–4, 8).

Components do not interact with each other directly. Their interaction is dependent on their membership in dynamic groups called ensembles. The key task of an ensemble is to periodically exchange knowledge between its coordinator and member components (determined by their roles). An ensemble instance is dynamically created/disbanded "around" its coordinator depending

```
1.  role Dockable:
2.    id, assignedDockingStationPosition
3.  role Cleaner:
4.    id, position, targetPosition, dirtinessMap
5.  role Dock:
6.    position, dockedRobots
7.
8.  component Robot features Dockable, Cleaner          /*the roles featured*/
9.    knowledge:
10.     id = . .
11.     position = { .. , ..}
12.     dirtinessMap = {}
13.     targetPosition = null
14.     assignedDockingStationsPosition = null
15.
16.   process move in mode Cleaning, Searching /* refer to mode-state machine of Fig. 8*/
17.     in targetPosition
18.     inout position
19.     inout dirtinessMap
20.     function:
21.       position ← move (targetPosition)
22.       dirtinessMap ← update(position, dirtinessMap)
23.     scheduling: periodic( 100ms )
24.
25.   process clean in mode Cleaning
26.     in position
27.     inout dirtinessMap
28.     function:
29.       if dirty(position):
30.         dirtinessMap ← clean(position, dirtinessMap)
31.     scheduling: periodic( 1000ms )
32.
33. /* similar spec for other processes of Robot */
34. /*  instatiation of component instances Robot1, Robot2, and Robot3 and setting of their
35.  initial  knolwedge is skipped for brevity*/
```

Fig. 2. Excerpt from the DSL of DEECo components of the cleaning robots example.

on which member components satisfy the membership condition in the ensemble specification. At design-time, an ensemble specification consists of (i) the roles that the member and coordinator components should feature, and (ii) a membership condition (Fig. 3, lines 11–17), and (iii) a knowledge exchange function, which specifies the knowledge exchange that takes place between the components in the ensemble (lines 18–19). For instance, the components featuring the Dockable role (e.g. Robot) and satisfying the membership condition (line 5) can form an ensemble with a component featuring the Dock role (i.e. with a DockingStation) to coordinate on the docking activity (lines 1–9).

The task of matching a component role and an ensemble role can be interpreted as establishing a connector in a classical component model; such a connector lasts only until the next evaluation of the membership condition (periodically triggered, line 20).

Self-Adaptation in DEECo. Overall, the dynamic grouping of components into ensembles determining their communication allows for software architecture that is dynamically (self-)adapted to the current components' knowledge values. The semantics of ensembles reflects the idea of MAPE-K (Monitor-Analyze-Plan-Execute and Knowledge-base) (Kephart and Chess, 2003): Consider an ensemble *E*. The membership condition MC of E is evaluated (analyzed) periodically, requiring systematic monitoring of the variables (knowledge parts) in all components featuring E's roles. From all components considered in a particular MC evaluation, only those satisfying MC are planned to be the members/coordinator of E. This plan is then executed and communication of the members/coordinator via knowledge exchange is realized.

```
1.  ensemble DockingInformationExchange:
2.    coordinator: Dock
3.    member: Dockable
4.    membership:
5.      coordinator.dockedRobots.size() <= 3
6.    knowledge exchange:
7.      coordinator.dockedRobots ← member.id
8.      member.assignedDockingStationPosition  ← coordinator.position
9.    scheduling: periodic( 1000ms )
10.
11. ensemble CleaningPlanExclusion:
12.   coordinator: Cleaner
13.   member: Cleaner
14.   membership:
15.     coordinator.targetPosition == member.targetPosition
16.       and distance(coordinator.position, coordinator.targetPosition)
17.             < distance(member.position, member.targetPosition)
18.   knowledge exchange:
19.     member.targetPosition ← null
20.   scheduling: periodic( 1000ms )
```

Fig. 3. Excerpt from the DSL of DEECo ensembles of the cleaning robots example (cont.).

Moreover, the components also self-adapt locally according to the state machines they are associated with: The semantics of

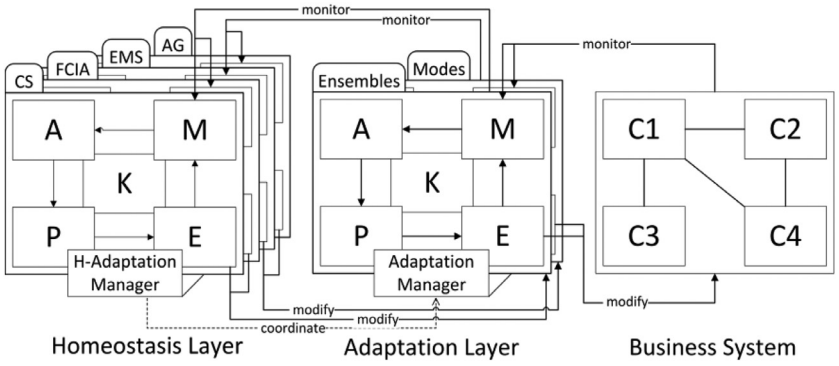


Fig. 4. Three-layered architecture with homeostasis layer.
The monitoring-oriented lines express data flow, while the other oriented lines control flow.

switching modes within a component also reflects the idea of the MAPE-K self-adaptation loop (Kephart and Chess, 2003): Consider a component C and its associated mode-state machine M_C. In M_C, the transition guards from the current state are periodically evaluated based on monitoring the variables (knowledge parts featured in the guards) of C; then it is analyzed which of the eligible transition should be selected so that the next mode is planned. Finally, the next mode is brought to action (executed).

To summarize, in DEECo, self-adaptation is performed by two mechanisms applied in parallel: (i) Mode-switching at the level of individual components, (ii) dynamic participation of components in ensembles. In principle, each instance of such a self-adaptation mechanism defines a particular self-adaptation strategy (in the sense of (Cheng et al., 2012)), being characterized in each component by a specific mode-state machine, and in each ensemble instance by a specific membership condition and knowledge exchange function. Technically, this is realized by an Adaptation manager (part of the run-time framework of DEECo (Bures et al., 2013)), which takes the specification of mode-state machines and ensembles as definition of self-adaptation strategies and invokes them accordingly.

3. Homeostasis at the architectural level

Our approach modifies/adds/removes self-adaptation strategies at run-time when the system requirements and/or the environment assumptions for which the strategies have been designed for are not met anymore. Our approach realizes this in an additional adaptation layer (*homeostasis layer*). Conceptually, the three layers presented in Fig. 4 follow the three-layered architecture for evolution of dynamically adaptive systems proposed by Perrouin et al. (2012). Contrary to their work, however, we do not use an evolution layer to switch between self-adaptation strategies. Instead, we propose the top layer to change the employed adaptation strategies by *homeostatic mechanisms* (*H-mechanisms)* based on a MAPE-K loop governed by an H-Adaptation Manager (Fig. 4).

To illustrate the concepts of the Homeostasis Layer, we present four H-mechanisms: Collaborative sensing, Faulty component isolation from adaptation, Enhancing mode switching, and Adjusting guards in mode switching. Each of them follows the MAPE-K loop. In its Monitor and Analysis phases, it monitors and analyzes specific exceptional situations in the Adaptation Layer (*triggers*) and reacts by activating its Plan and Execute phases, which, in turn, modify a self-adaptation strategy at the Adaptation Layer. Here, to avoid conflicts, the H-Adaptation Manager coordinates the Plan and Execute phases of H-mechanisms with the Adaptation Manager which controls the application of the self-adaptation strategies. This is enabled by storing information on the currently running adaptation strategies as well as on which adaptation strategy specifications are locked in the Knowledge-base of the Adaptation Manager (further details to be found in Section 3.5). It should be emphasized that the Adaptation manager and the H-Adaptation manager are conceptually singletons. Nevertheless, their functionality can be implemented both in a centralized and decentralized way in the run-time framework.

In principle, the Homeostasis Layer could be avoided by enhancing the Adaptation Layer to handle all the exceptional situations; however, this would make their specification clumsy and error prone. Therefore, we hoist the handling of these exceptional situations to the architectural level and modify the Adaptation Layer by the Homeostasis Layer at run-time. Moreover, the adoption of such an architecture style provides more design flexibility by allowing incremental tuning up of the Adaptation Layer.

As a reference implementation, the Adaptation Layer in our running example is built upon the self-adaptation mechanisms in DEECo, by modifying/adding/removing the self-adaptation strategies defined by mode-state machines' and ensembles' instances at run-time. Technically the Adaptation manager and the H-Adaptation manager are both implemented in a centralized way. This reference implementation is available online.[2]

To support reuse of the proposed H-mechanisms in other component models than DEECo, we specify the recommended Adaptation Layer interfaces that a component model should provide to apply the H-mechanism along with the description of each H-mechanism in the following subsections. In general, such a component model should support self-adaptation through (i) dynamic connectors and/or (ii) mode switching. By dynamic connector we understand a connector established/removed dynamically between components based on matching their available ports. In the DEECo reference implementation, the concept of connector is realized as an ensemble, and the concept of port as role. The exact semantics of "matching" and "available" is specific to a particular component model. Typically, matching is based on the port name and/or its type. Furthermore, it may be further constrained by a filtering predicate upon the component data fields. Examples of component models that support dynamic connectors include (Bures et al., 2013; Hall et al., 2011; Escoffier et al., 2007; Srinivasan and Mycroft, 2008; Wolfinger, 2012). To address the diversity in the actual semantics, we specify below the minimal functionality the Knowledge-base of the Adaptation Layer has to offer to support the particular H-mechanism—we do so by providing a conceptual model featuring recommended interfaces and also indicating which of the component/connector/mode related entities

[2] https://github.com/d3scomp/uncertain-architectures.

are assumed to be realized in the Adaptation Layer when the H-mechanism is considered for application.

3.1. H-Mechanism #1: Collaborative sensing (CS)

sasiCPS are often large data-intensive systems with components that perform sensing of physical properties via hardware sensors (e.g. GPS, accelerometer, thermometer) with various reliability margins. When components rely on sensor readings to satisfy important functional requirements (e.g. a robot needs to know its position to plan its path to a destination), it becomes extremely important to deal with sensor malfunctioning to still enable environment sensing at run-time. For illustration, consider the situation where the downwards-looking camera of a robot *R* in our running example starts failing and consequently R loses the ability to detect dirtiness on the floor (and, to update its dirtinessMap).

A way to overcome the problem of sensor malfunctioning is to take advantage of the data dependencies and redundancies that may exist in sasiCPS due to components sensing the same or a similar property *Q*. Collaborative sensing (CS) H-mechanism provides an adequate approximation of property *Q* for a faulty component. CS is based on defining a new self-adaptation strategy on the fly—technically, in DEECo, by creating an additional ensemble specification with knowledge exchange function providing the desired approximation.

CS involves two key computational steps in its Analysis and Plan phases: (i) *CS Analysis*—identification of data dependencies and (ii) *CS Plan*—approximation of *Q*. While CS Plan is relatively easy to realize once a dependency relation is identified, for the two main tasks of CS Analysis, namely for the *data collection* and the *acquiring of dependency relation*, there are different options. The data collection can be done on a real system or on a simulated one. Acquiring the dependency relation can be accomplished via statistical or machine learning techniques.

For illustration, in the following we assume that the data collection is performed on a real system and that the dependency relations are created by applying the following statistical method: Let us assume that in its Monitor phase CS collects the values of preselected knowledge fields of the set of components of the same type in the latest time instances $t = 1..n$. Furthermore, for acquiring the dependency relation, CS Analysis checks all the aggregated knowledge to find out which knowledge fields are dependent on others. Let $C_i.k_l^t$ be the value of knowledge field k_l of component C_i at a time instance t, and $\{C_i.k_l\}_1^n$ denote the time series of the knowledge field k_l of component C_i at time instances 1 to n. Further let $\mu_{k_l}(C_i.k_l^t,\ C_j.k_l^t)$ be the distance between two knowledge values of k_l^t in components C_i and C_j measured by metric μ specific to k_l. Then, for all component pairs $C_i,\ C_j;\ i \neq j$, having the fields k_l and k_m, CS Analysis computes the boundary Δ_{kl} such that the implication $\mu_{k_l}(C_i.k_l^t, C_j.k_l^t) < \Delta_{k_l} \Rightarrow \mu_{k_m}(C_i.k_m^t, C_j.k_m^t) < T_{k_m}$ for the time instances $t = 1..n$ is satisfied in (at least) the specified percentage of all the cases (confidence level a_{k_m}, e.g. 90%). Here T_{k_m} represents the tolerable distance threshold and is provided for each k_m. The CS Analysis concludes that the value of $C_i.k_m^t$ is close to the value of $C_j.k_m^t$ (and vice versa) for t such that the values of $C_i.k_l^t$ and $C_j.k_l^t$ are close as well.

Thus, when a component C_f fails to sense the values of k_m, an approximation of this property has to take place. This is done by CS Plan by creating an ensemble with the exchange function $C_f.k_m := C_j.k_m$ and membership condition $\mu_{k_l}(C_f.k_l, C_j.k_l) < \Delta_{k_l}$. If more than one C_j satisfies the membership condition, an arbitrary one is selected. The ensemble is deployed and started by the E phase of CS.

The task to compute the boundary Δ_{k_l} is resource and time demanding but there are techniques that can lower the time needed to finish, such as sorting the data according to $\mu_{k_m}(C_i.k_m^t, C_j.k_m^t)$ or using sampling of the gathered data to obtain a statistically significant answer. There are of course a number of other methods to detect dependencies between data such as linear regression, k-nearest neighbors, neural networks, etc.

```
1.  role DirtinessMapRole:
2.      position, dirtinessMap
3.
4.  ensemble DirtinessMapExchange:
5.      coordinator: DirtinessMapRole
6.      member: DirtinessMapRole
7.      membership:
8.          // Member and coordinator must be "close" to form the ensemble
9.          // The robot with broken sensor becames the coordinator
10.         close(coordinator.position, member.position)
11.          and obsolete(coordinator.dirtinessMap)
12.     knowledge exchange:
13.         coordinator.dirtinessMap ← member.dirtinessMap
14.     scheduling: periodic( 1000ms )
```

Fig. 5. DSL excerpt from ensemble specification generated when applying CS for the cleaning robots.

Coming back to our running example, the situation where the camera of robot R starts failing, is a trigger of the Plan and Execute phase of the CS H-mechanism which will create a `DirtinessMapExchange` ensemble, the membership condition of which states that R becomes the coordinator and the other robots that are closer to R than the given threshold (obviously, when their positions are close, their maps are "close") become its members. By knowledge exchange, R adopts the `dirtinessMap` of the closest member (Fig. 5) and can resume its cleaning operation.

Assumptions on the Adaptation Layer. In Fig. 6, the conceptual model defines the interfaces (and other entities) the Knowledge-based of the Adaptation Layer should provide to support CS. Considering again the robot R as above, the key required functionality of the Adaptation Layer can be illustrated as follows: CS via `getFaultyKnowledge` of `IComponent` learns about faults in R. When failing of the dirt sensor is observed, via IConnector Manager, CS creates a new connector specification C having a `consumer` port and `producer` port. Furthermore, a new `consumer` port is added to R (`addPort` of `IComponent`) and new `producer` ports are added to other components with relevant knowledge structure.

Thanks to the `filter` predicate specified in `addConnector`, C instances are dynamically attached to R and the other components determined by the value of `filter`. This predicate reflects the condition $\mu_{k_l}(C_i.k_l^t, C_j.k_l^t) < \Delta_{k_l}$ as derived by the CS Analysis. To collect data for the analysis, CS relies on the `getKnowledge` interface of `IComponent`.

Computation Complexity. For the complexity analysis of this and the other mechanisms, we assume that the computational complexity of Monitor and Execute phases is *O(1)* since they involve a constant number of elementary operations for each run (i.e. no iterative computations), and only examine the Analysis and Plan phases.

CS Analysis has alternative subtasks. The above-described algorithm investigates all potential relations between knowledge fields to identify dependencies that determine ensembles in the CS Plan phase. Assuming the algorithm considers a time window of length n and m knowledge fields, the complexity of applying the algorithm is $O(c^2m^2n)$, where c is the number of components. The CS Plan phase has a complexity of O(1), since it simply takes the result of the Analysis and creates ensemble/connector specifications.

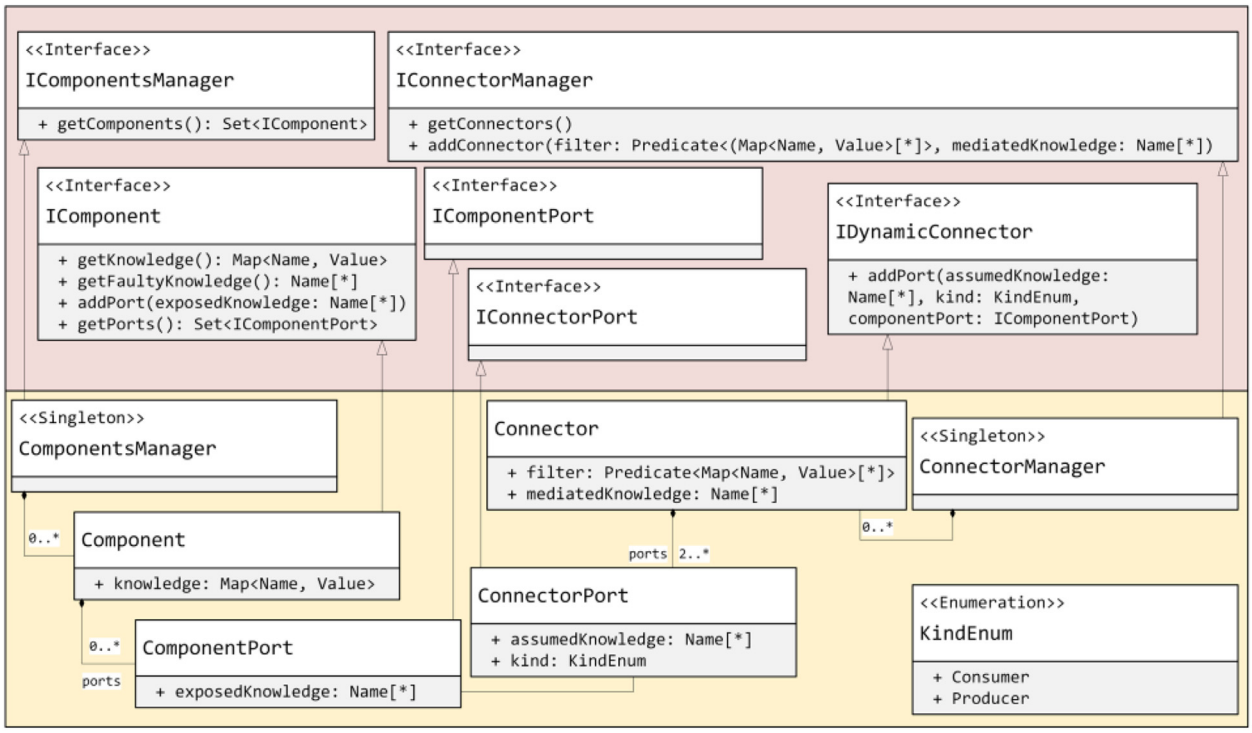


Fig. 6. Collaborative sensing: conceptual interfaces for the adaptation layer knowledge-base.

3.2. H-Mechanism #2: Faulty component isolation from adaptation (FCIA)

The idea of the faulty component isolation from adaptation (FCIA) H-mechanism is rooted in the well-known fault-tolerance mechanism: When a component starts malfunctioning it has to be isolated from the rest of the system and its activity taken over by another non-faulty component providing the same functionality. In essence, FCIA addresses the situation where a component *A* starts emitting faulty values of its property *Q*. The identification of such a situation can be based either on learnt fault models or on empirical knowledge about the values that can be considered faulty. In such a case, FCIA modifies the adaptation strategies that count on *Q* in order to prevent the "contamination" of other components with faulty values of *Q*.

For illustration, consider a situation in the running example where a docking station *DS* is not able of having docked robots charge anymore due to some error or malfunction, while still being advertised as operational to the robots that are technically members of the `DockingInformationExchange` ensemble associated with *DS* (Fig. 3). As a result, a Robot may still queue at the faulty *DS*. This is a trigger for applying the Plan and Execute phases of FCIA H-mechanism. In essence, FCIA modifies the `DockingInformationExchange` specification in such a way that *DS* is excluded from being the coordinator of one of its instances. Technically, this can be done by modifying the membership condition to make it not satisfiable for *DS*.

Assumptions on the Adaptation Layer. In Fig. 7, the conceptual model defines the interfaces (and other entities) that the Knowledge-base of Adaptation Layer should provide to support FCIA. Considering again the faulty docking station DS as above, the key required functionality of the Adaptation Layer can be illustrated as follows: Via `getComponents` of `IComponents Manager`, FCIA gets access to the representation of the components in the Knowledge-base of the Adaptation Layer so that through monitoring (via `getFaultyKnowledge` of `IComponent`) it can learn about faults in DS. Furthermore, it identifies (via `getExposedKnowledge` of `IComponentPort`) the port(s) of DS to be removed (by `removePort` of `IComponent`) to isolate DS. Then, thanks to the dynamic connector principle, the affected Robots are connected to available Docking Stations.

Computation Complexity: In its Analysis phase FCIA traverses all components and obtains the list of faulty knowledge fields. The complexity of the traversal is bounded by $O(ck)$, where c is the number of components and k is the number of faulty knowledge fields per component. The FCIA Plan phase has a complexity of O(1), since it simply interprets the Analysis results to remove ports (or roles in DEECo) if needed.

3.3. H-Mechanism #3: Enhancing mode switching (EMS)

The motivation behind the enhancing mode switching (EMS) H-mechanism are cases where the behavior of a component specified by its mode-state machine is over-constrained. For illustration, consider the exceptional situation where there are far more Robots than DockingStations. Assuming similar energy depletion and similar initial energy budgets, if all Robots follow the mode-state machine depicted in Fig. 8, they might all switch to Charging mode at similar points in time when, for example, their energy falls below 20%. This would result in an increase of the average charging time caused by the need of queueing up at the docking stations.

Thus, instead of being stuck in situations that have not been anticipated at design time, it can be beneficial to relax the constraints and enlarge the space of actions that can be tried out to handle such situations. Building on this idea, the EMS H-mechanism adjusts the self-adaptation strategy implemented as a mode-state machine associated with a particular component type. In general, EMS monitors the values of the utility function associated with the given component type (Monitor and Analysis phases of EMS) and triggers the adaptation (Plan and Execute phases of EMS) when the average utility value of all instances of the given type is low. In essence, EMS systematically creates new transitions with the guard

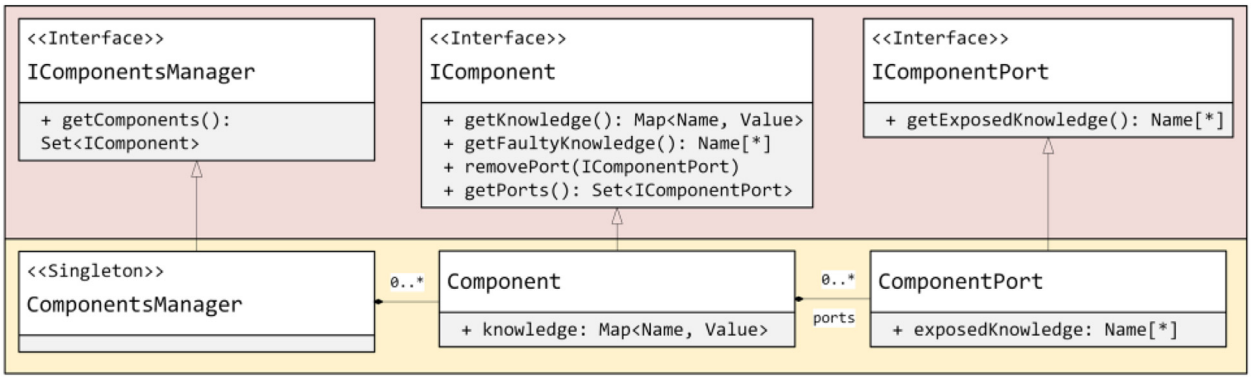


Fig. 7. Faulty component isolation from adaptation: Conceptual interfaces for the adaptation layer knowledge-base.

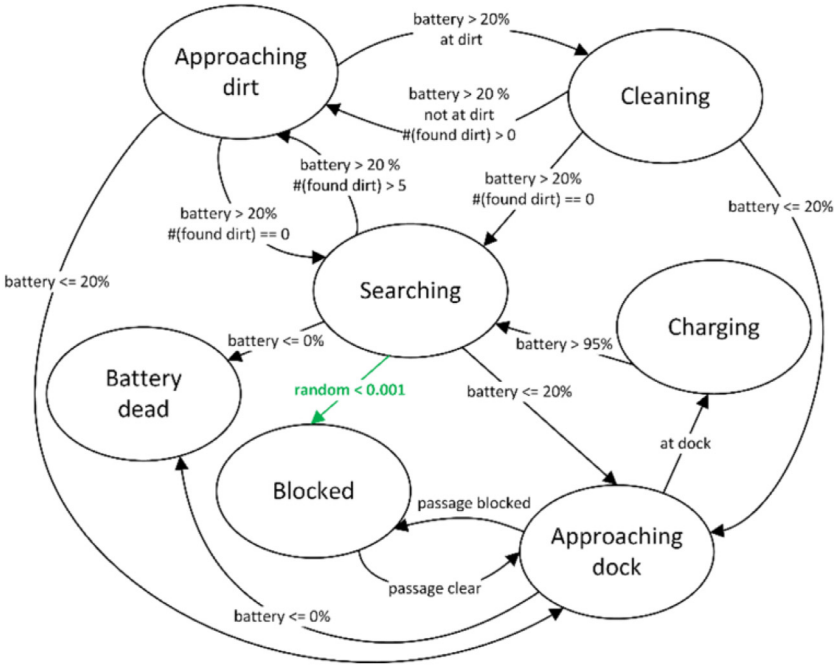


Fig. 8. Mode-state machine capturing the mode switching logic of the Robot component. Each state (mode) is associated with several processes. Transitions are guarded by conditions upon the Robot's knowledge. Changes introduced by the EMS H-mechanism are marked in (bold) green – added transitions are guarded by a probabilistic condition. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

of the form "random $< p$" among the modes where none of the original transition is present. These *randomized transitions* are to be applied with a higher priority than the original transitions and their effect is evaluated based on the utility function defined for the related component type. For simplicity, we consider the following very basic strategy of adding randomized transitions one by one and observing in a given time period the effect of each of them in isolation. In the end, the one that improved the average utility the most (if any) is kept in the mode-state machine (recall that it is applied for all component instances of the given type) and the Execute phase of EMS ends.

Coming back to our running example, the situation when the queuing and consequently charging time of robots takes longer than usual will act as a trigger for the Plan and Execute phase of EMS, because the average utility is affected. The utility U is computed as $U = E(t_{dirt\_appeared} - t_{dirt\_cleaned})$, where E is the expected value, $t_{dirt\_appeared}$ is the time when a dirt d appeared and $t_{dirt\_cleaned}$ is the time when d was cleaned. The EMS will change the mode-state machine of all robots by adding new transitions (depicted in green in Fig. 8). The new randomized transitions have the guard of the form random $< p$, for some probability p. The value of p depends on the frequency of mode-state machine evaluation of transitions and on the required level of manifestation of non-deterministic transitions. In our case, we picked $p = 0.001$ to observe the manifestation of non-deterministic transitions.

Assumptions on the Adaptation Layer. In Fig. 9, the conceptual model defines the interfaces (and other entities) that the Knowledge-base of the Adaptation Layer should provide to support EMS. Considering again the situation with far more robots than docking stations as above, the key required functionality of the Adaptation Layer can be illustrated as follows: EMS via `getComponents` of `IComponentsManager` gets access to the representation of components from the Knowledge-base of the Adaptation Layer, so that through monitoring (via `getAverageUtility` of `IComponentType`) it can observe the average decrease of the utility of robots (this is evaluated by `getUtilityThreshold` of `IComponentType`). Furthermore, it identifies (via `getTransitions` of `IModeChart`) all the transitions present in the mode-state machine and augments it with new (non-deterministic) transitions, one at a time, all of them with the same guard of the form "random $< p$" and of the same higher priority than of those existing so far. This is done by

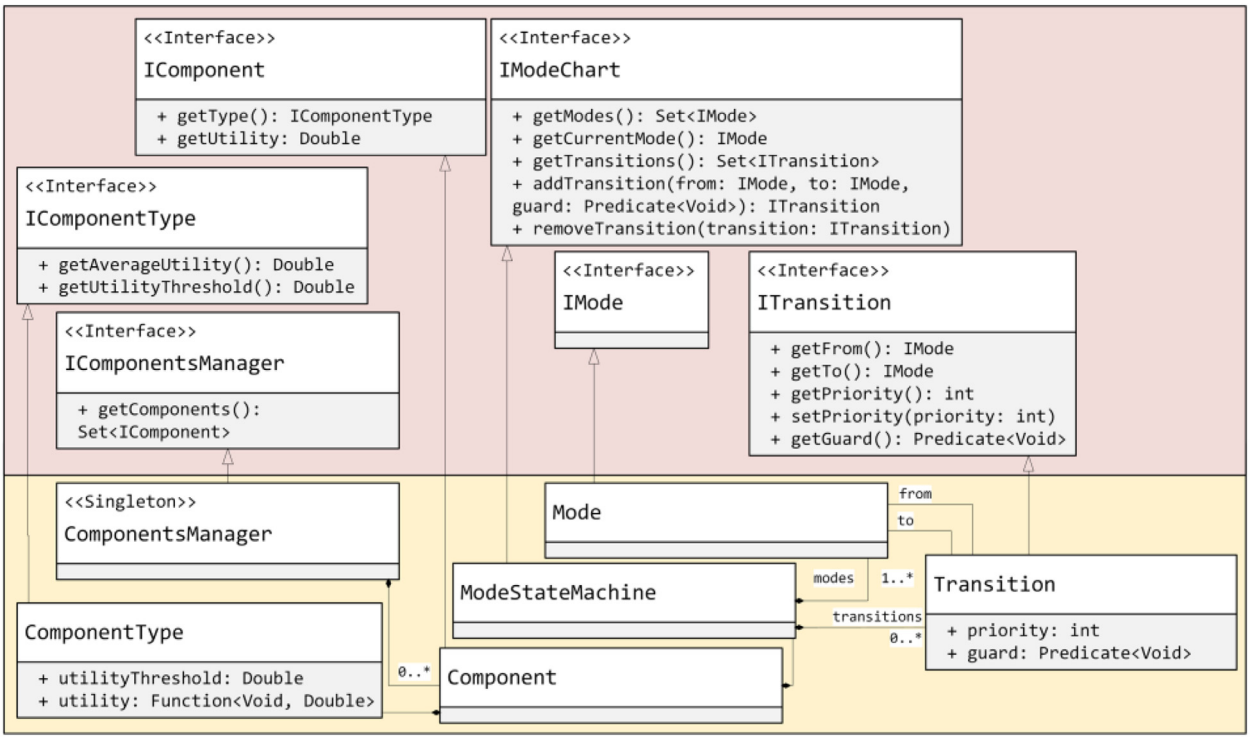


Fig. 9. Enhancing mode switching: conceptual interfaces of the adaptation layer knowledge base.

`addTransition` of `IModeChart`. Then, periodically, the average utility of robots is reevaluated and the added transition is kept only if it has improved the utility and removed otherwise. At each step of the local optimization algorithm (we used gradient descent (Snyman, 2005)), a new transition is tried out. As soon as the measured utility climbs above the specified threshold or a preset number of iterations is reached, the EMS mechanism terminates its Execute phase and the H-Adaptation Manager regains the control.

Computation Complexity. The EMS Analysis phase has a complexity of *O(1)*, since it is only concerned with calculating the results of the utility function, the complexity of which we assume to be *O(1)*. The EMS Plan phase adds, modifies or removes probabilistic transitions. This generally requires traversing the list of transitions in the component's mode-state machine. Assuming the component has t transitions, the complexity of EMS Plan is bounded by $O(tc)$, where c is the number of components.

3.4. H-Mechanism #4: Adjusting guards in mode switching (AG)

The idea behind Adjusting Guards in Mode Switching (AG) H-mechanism is that conditions in mode switch guards are sometimes unnecessarily strict. This happens due to the semantics of the mode-state machine that requires the guards to be logical expressions, often captured as hard constraints involving constants which may be hard to set at design time. For illustration, consider the transition `Searching` → `Approaching Dock` (Fig. 8) guarded by the constraint $energy \leq 20\%$. In many cases (including this one) this does not mean that 19% would be too little or that 21% would be too much, rather it is an educated guess due to lack of experimental data which would give statistical confidence.

As a remedy, the AG H-mechanism tries to improve the performance of the system by optimizing the constants in the mode switch guards. Similar to EMS, AG monitors the values of the utility function associated with the given component type and triggers the adaptation (its Plan and Execute phases) when the average utility value across all instances of the given type is low. In such a case, AG systematically adjusts the constants and observes the effect on the average utility. This process is driven by a local optimization algorithm (we used gradient descent) that selects the respective constant and decides in which direction and how much it should be tuned. In essence, compared to EMS which addresses the situation when the system is being stuck due to a situation not anticipated in the system design, AG aims at optimizing the efficiency of system functionality at runtime.

Coming back to our running example, the AG H-mechanism systematically observes the swiftness of the cleaning robots in removing the dirt, which is expressed by the same utility function as in the EMS H-mechanism. Since the utility function in the example is constructed as the average time from discovering the dirt to cleaning it, the statistical confidence about the utility is directly proportional to the number of robots involved, the rate of dirt appearance, and the time interval over which the utility function is observed. Once AG reaches sufficient statistical confidence about the effect of setting a particular guard constant, it proceeds to the next guard or with another value for the same guard.

Assumptions on the Adaptation Layer. In Fig. 10, the conceptual model of AG defines the interfaces (and other entities) the Knowledge-base of the Adaptation Layer should provide to support this H-mechanism. In many respects, the conceptual model is similar to the one of EMS—there is the ability to inspect components and their mode charts and to query the utility function. What is extra compared to EMS is the ability to introspect the guard, in particular to query and set the tunable constants in the guard (via `getGuardParams` and `setGuardParams` of `ITransition`).

Computation Complexity. Similar to EMS, the AG Analysis phase is assumed to have a complexity of *O(1)*—the complexity of utility function calculation. The AG Plan phase has to traverse the guards in each component's state machine and inside the guards it traverses the settable parameters. The overall complexity of this

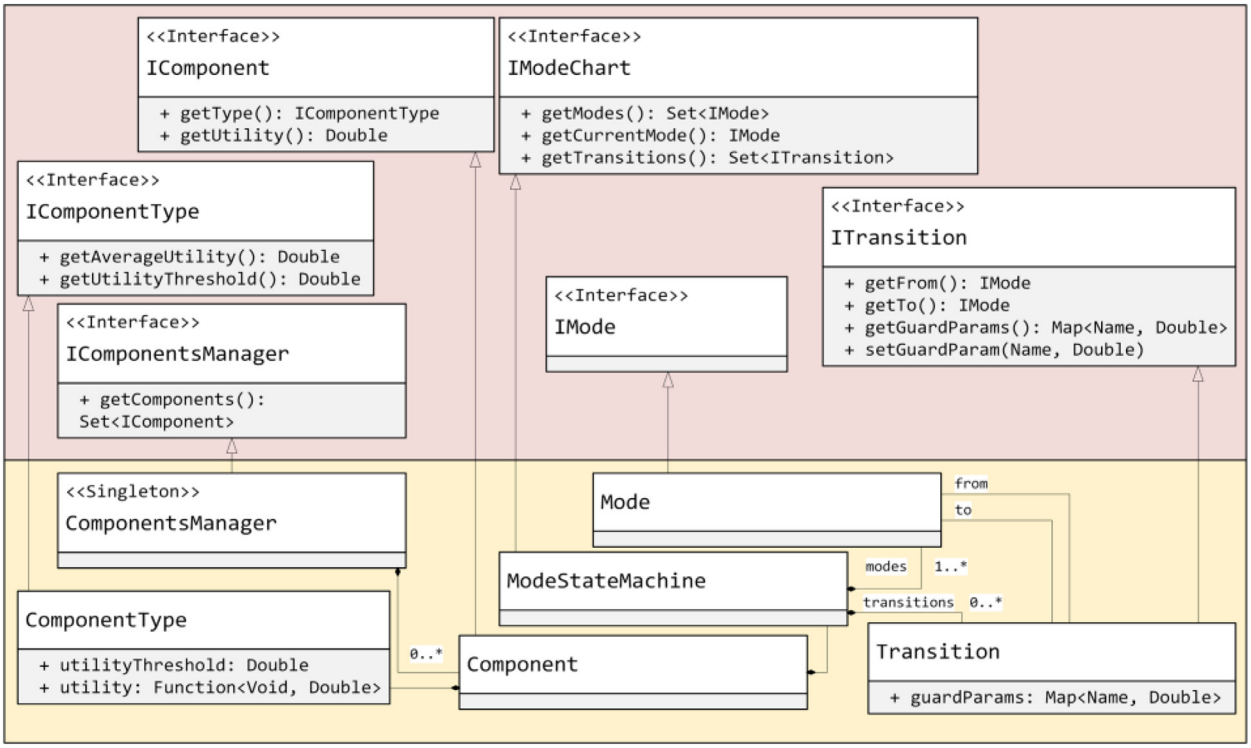


Fig. 10. Adjusting guards in mode switching: conceptual interfaces of the adaptation layer knowledge-base.

phase is thus bounded by $O(tpc)$, where t is the maximum number of transitions per component, p is the maximum number parameters per transitions, and c is the number of components.

The difference between EMS and AG should be emphasized: EMS is to be applied when the system is stuck in a situation not anticipated at the design time, while AG is intended to remedy an inefficient performance of the system. Even though in both cases the triggering conditions are based on the utility function, the actual values of the function to be tested are different (and so are the adaptation strategies).

3.5. The whole picture

In principle, the Monitor and Analysis phases of all four H-mechanisms (recall Fig. 4) are activated periodically (the period is application-specific) and running in parallel. The trigger of each H-mechanism, which signals to proceed with the corresponding Plan and Execute phases, is evaluated at the end of the Analysis phase. If it evaluates to true, the H-Adaptation Manager steps in to coordinate potential conflicts of multiple H-mechanisms in their Plan and/or Execute phases. Moreover, updating an adaptation strategy in the Adaptation layer while it is being executed should obviously not be possible in general. Avoiding such potential conflicts is another task of the H-Adaptation Manager which has to cooperate with the Adaptation manager, asking for its permission, to fulfill the task. In general, such multi-layer controller coordination is a hard problem, especially if possible parallelization of control activities is considered.

Below, we provide a description of the coordination logic we employed in our experiments. It stems from the simple assumption that triggers of the Plan and Execute phases of the H-mechanisms are indicators of exceptional situations. Since they appear "exceptionally", is it not a big harm to handle the case that several triggers appear simultaneously in a simplistic way:

H-mechanism calling H-Adaptation manager. Let us assume that a run of the Monitor and Analysis phases of an H-mechanism m ends up with a trigger t. Then m asks the H-Adaptation manager for a permission to continue with the Plan and Execute phases to handle t. If the permission is granted, the H-Adaptation manager refuses any other trigger t' issued by an H-mechanism m' until handling of t is finished. In principle, this means that the "unsuccessful" trigger can be potentially re-generated by a future periodic run of the Monitor and Analysis phases of m'. This does not have to be necessarily the case, because the triggering condition of t' does not have to be valid anymore as a consequence of handling t. In the very special case that multiple triggers appear at the same time, which was not possible—by design—in our experiments, since the H-mechanisms had different, non-overlapping triggers, one of them is randomly selected by the H-Adaptation manager to be handled and the rest of them are refused. A priority-based mechanism could also be introduced to give precedence to certain H-mechanisms by design. In our experiments, there was no need for prioritization, since the H-mechanisms we employed could not be triggered together. Therefore, we did not consider prioritization mechanisms in our work.

Cooperation between the H-Adaptation manager and the Adaptation manager. Assume again that m asks the H-Adaptation manager for a permission to continue with the Plan and Execute phases to handle t. To grant the permission, the H-Adaptation manager calls the Adaptation Manager asking for activating a lock l of the data structures to be modified by handling t (i.e. locking the specifications of such adaptation strategies in the first place). Naturally, the call does not return until l has been successful (a potential wait would be due to another lock by the adaptation being currently executed in the adaptation layer). The lock l remains active until the handling of t is finished (i.e. the phases Plan and Execute of m are completed). Then unlocking of l is carried out by another call of the Adaptation manager by the H-Adaptation manager. This concludes handling of t.

```
1.  ensemble RefillStationEnsemble:
2.    coordinator: RefillProvider  # can be either a Hydrant or RefugeeCamp component
3.    member: RefillConsumer    # a FireBrigade component
4.    membership:
5.      distance(coordinator.position, member.position)
6.            < distance(member.refillTarget, member.position)
7.    knowledge exchange:
8.      member.refillTarget  ← coordinator.position
9.    scheduling: periodic( 1000ms )
10.
11. ensemble TargetFireZoneExchange:
12.   coordinator: FireFighter  # a FireBrigade component
13.   member: FireFighter       # a FireBrigade component
14.   membership:
15.     coordinator.helpTarget == null and coordinator.extinguishing and !member.refilling &&
16.       and distance(coordinator.position, member.position)
17.             < distance(member.helpTarget, member.position)
18.   knowledge exchange:
19.       member.helpTarget ← coordinator.position
20.       coordinator.helpingFireFighter ← member.id
21.   scheduling: periodic( 1000ms )
```

Fig. 11. DEECo ensembles of the emergency response system.

4. Experimental evaluation

In this section, we present an experimental evaluation of the feasibility and effectiveness of the H-mechanisms. In particular, our experiments aim at answering our research question (a): To what extent can each H-mechanism repair the problem it is intended to repair? The evaluation is based on experimental results obtained by implementing the H-mechanisms in JDEECo, a Java implementation of the DEECo component model (Bures et al., 2013). We implemented each H-mechanism as a plugin to the JDEECo framework, taking advantage of its modeling and simulation capabilities. In our implementation, all H-mechanisms[3] are governed by a centralized H-Adaptation Manager implemented as an isolated DEECo component.

To answer the research question, we performed an experimental study with two simulated systems acting as testbeds. The first testbed is an implementation of the running example of cleaning robots already presented in Section 2.1, while the second is an emergency response system described next.

4.1. Second Testbed: Emergency response system

Our second testbed is based on the RoboCup Rescue Simulation[4] (RCRS), a research and educational project targeted towards evaluation of multi-agent solutions in disaster response scenarios. RCRS provides a simulation platform that imitates a city after an earthquake. The platform provides a city map which includes streets, intersections, buildings, and stationary and platoon agents. In our testbed, we concentrated on the fire-fighting scenarios and considered *Hydrants* and *Refugee Station* as stationary agents and *Fire Brigades* as platoon agents. Additionally, RCRS provides other types of agents such as *Ambulance Centers* and *Ambulance Teams*. *Fire Brigades* are responsible for extinguishing buildings on fire. They can only carry a limited amount of water and need to navigate to the closest *Hydrant* or *Refugee Station* to refill their water tanks. They need to coordinate with each other to expedite the fire extinguishing process.

In the DEECo model of the disaster response scenarios, the RCRS agents correspond to DEECo components. Communication between components is prescribed by two ensembles listed in Fig. 11, which allow for `Fire Brigades` to find a `Hydrant` or `Refugee Station` to refill (`RefillStationEnsemble`) and to form teams of collaborating `FireBrigades` (`TargetFireZoneExchange`).

4.2. Experimental study

The cleaning robots testbed (A) was implemented in JDEECo, while testbed (B) was implemented in the RCRS simulation platform. For each testbed, we designed and conducted targeted controlled experiments consisting of multiple simulation runs of predefined scenarios, each being a combination of deliberatively introduced faults to be addressed by a particular H-mechanism.

In total, we considered eight different scenarios for each testbed, which are described in Fig. 12. To measure the overall utility of a system run, we used the following metrics:

(A) Cleaning robots: The mean of computed time required for a tile that got dirty until it is cleaned. Smaller values are better.
(B) Emergency response: The RCRS built-in score function associated with the fraction of buildings not damaged by fire. Larger values are better.

For the sake of statistical significance, each scenario was run in 100 iterations. The reported values for the above metrics were averaged across these iterations.

4.3. Results and interpretation

CS and FCIA. Fig. 13 shows the values of the overall system utility of scenarios 1–5 in the form of boxplot diagrams where the number associated with the red line denotes the median of the sample.

Scenarios A.1 and B.1 represent the vanilla cases (no faults—no H-mechanism activated), acting as the baselines. Not surprisingly, in the other scenarios the overall utility decreases when a fault is introduced but no H-mechanism is triggered (scenarios A.2, A.4 and B.2, B.4). When the respective H-mechanism is triggered (scenarios A.3, A.5 and B.3, B.5), the overall utility improves. In scenarios A.3 and A.5, the overall utility does not reach the baseline scenario (A.1). However, in scenarios B.3 and B.5, the overall utility in the corresponding scenarios even exceeds that of the baseline

[3] Available at: https://github.com/d3scomp/uncertain-architectures.
[4] http://roborescue.sourceforge.net/.

Scenario	Fault	Mechanism	Number of docking stations
A.1	-	-	3
A.2	A robot's dirtiness sensor malfunctions	-	3
A.3	A robot's dirtiness sensor malfunctions	CS	3
A.4	A docking station emits wrong availability data	-	3
A.5	A docking station emits wrong availability data	FCIA	3
A.6	Too many robots w.r.t. docking stations	-	1
A.7	Too many robots w.r.t. docking stations	EMS	1
A.8	Too many robots w.r.t. docking stations	AG	1

(A) Cleaning Robots: Simulation duration 600 s, environment size i20 x 20, number of robots 4.

Scenario	Fault	Mechanism
B.1	-	-
B.2	2 firefighters can't detect burning buildings	-
B.3	2 firefighters can't detect burning buildings	CS
B.4	12 (randomly chosen) out of 18 hydrants are not working	-
B.5	12 (randomly chosen) out of 18 hydrants are not working	FCIA
B.6	deployed system suboptimal	-
B.7	deployed system suboptimal	EMS
B.8	deployed system suboptimal	AG

(B) Emergency Response: Simulation duration 600 steps, number of fire fighters 12.

Fig. 12. Scenarios considered in the controlled experiments.

(B.1). We provide the reasons for this counter-intuitive result in the next paragraphs.

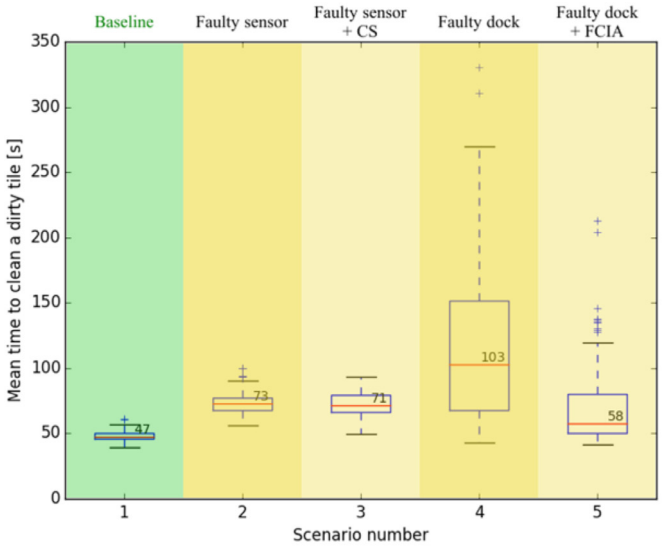
When applying CS to testbed A (scenario A.3), a dependency relation (Section 3.1) was identified based on the observation that the closeness of the positions of `Robot` components implied similar values in their `dirtinessMaps`. This resulted in the creation and deployment of a `DirtinessMapExchange` ensemble (Fig. 5). The associated metrics, tolerable distances, and confidence levels are depicted in Figs. 14 and 15. Similarly, when applying the same H-mechanism (i.e. CS) in testbed B (scenario B.3), a dependency relation was identified based on the observation that the closeness of fire fighter's positions implied similar values in their `burningBuildings` knowledge. This resulted in the creation of an ensemble specification that allowed nearby fire fighters to exchange information about burning buildings (Appendix A). Scenario B.3 outperforms the baseline (B.1) because the distance of nearby firefighters in the generated ensemble is larger than the distance firefighters can detect fire.

When applying FCIA to the two testbeds, the utility in both cases is improved since the roles of the faulty components are removed. Scenario B.5 even outperforms the baseline. The reason for this is application-specific: when nearby Hydrants are not available any more for fire fighters to refill their water tanks, refilling happens at nearby Refugee Stations (the second choice of fire fighters). Since refilling happens faster in Refugee Stations than in Hydrants (by design in the RCRS simulator), the overall utility increases w.r.t. the baseline.

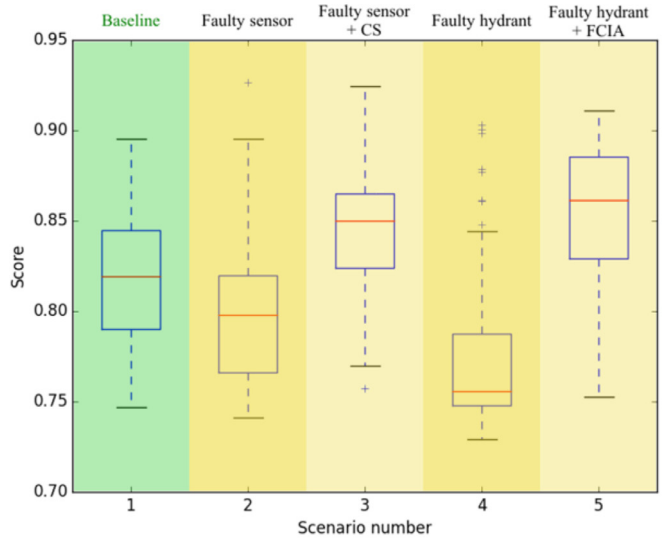
In the rest of the section, we describe only the results of the application of EMS and AG to testbed (A). When applying EMS and AG to testbed (B), we obtained similar results. We refer the interested reader to Appendix A where the results obtained while experimenting with all four H-mechanisms in testbed (B) are provided.

EMS and AG. Scenario 6 serves as a baseline for both EMS and AG. The effect of EMS is illustrated in Figs. 16 and 18. There, the compared effect of an added non-deterministic transition is compared to the baseline. In this case, the baseline consists of 4 robots and 1 docking station. Fig. 16 shows the utilities of the system when one non-deterministic transition is added at a time. We can see that some transitions improve the utility of the system and some worsen it. We can see that when the transition from Searching mode to Blocked mode was added, the system utility slightly improved. This is intuitive, because when the robot is in Blocked mode and is not blocked by another robot, it switches to the Approaching dock mode and starts charging when reaching the dock. Since there are too many robots for only one docking station, it helps the system when robots go to charge more randomly and do not queue as much in the docking station. Fig. 18 shows the utilities when two transitions, the ones that yield the best utilities in Fig. 16, are added at a time. When EMS is applied in scenario 7, it learns the utilities that the system yields when particular transitions are added. After adapting the system, due to the introduced probabilistic mode switching (probability of 0.001 was used), the robots started visiting the docking station at different times. Hence, the overall queueing time was reduced and the overall utility slightly increased. EMS needs time to auto-calibrate (performed in training simulations for each transition) as it searches for the best transition that improves the system utility.

Scenario 8 is trained in the same way as scenario 7. Fig. 20 shows the system utilities when different guards are modified, one at a time. Fig. 22 shows the utilities of the system when a combination of two guards is adjusted. AG learns from these measured values which guards to adjust and how to adjust for improving the running system. As observed, with AG robots charge more often and do not spend long time charging. Therefore, the utility



(A) Scenarios A1 – A5. Smaller values are better.



(B) Scenarios B1 – B5. Larger values are better.

Fig. 13. Overall system utilities.

improves. Additionally, AG helps when robots start cleaning sooner as they find less dirt than in the baseline.

5. Discussion

By employing two distinct architectural layers—"standard" self-adaptation and adaptation of self-adaptation strategies (the task of H-mechanisms), our solution basically follows the principle of architectural hoisting (Fairbanks, 2014)—separating concerns by assigning the possibility for a global system property (here self-adaptation) to system architecture. Even though the H-mechanisms layer can be interpreted as (high-level) exception handling in self-adaptation settings and can be implemented at the same level of abstraction as the self-adaptation itself, achieving the same functionality without the H-mechanism layer would make the code of ensembles and components very clumsy. Architectural hoisting makes the separation of these concerns much easier and elegant.

```
1.  def dirtinessMapDistance(map1, map2):
2.    dist = 0
3.    // for each node in the global map, if visited in
4.    // same-ish time add penalty if needed
5.    for n in DirtinessMap.getNodes():
6.     if(map1.getVisited().get(n) -
7.          map2.getVisited().get(n) <= timeWindow):
8.       dirt1 = map1.getDirtinessIn(n)
9.       dirt2 = map2.getDirtinessIn(n)
10.      if(dirt1 - dirt2 > dirtWindow):
11.       dist = dist + differencePenalty
12.   return dist
```

Fig. 15. Distance metric for `dirtinessMap` field of `Robot` component.

It should be emphasized that learning is an inherent part of the proposed mechanisms. CS builds data correlation models adhering to certain parameters, namely tolerable distances and confidence levels. FCIA builds on (ideally) learnt models of faults. EMS and AG learn the response (utility) of the system when changing the mode-state chart of certain components.

We return here to answering the research questions articulated in Section 1 and conclude this section by mentioning the limitations of our approach.

To what extent can each H-mechanism repair the problem it is intended to repair? Our experimental results provide evidence that each H-mechanism is indeed solving the problem it is intended to solve. The extent to which the problem is repaired depends on the application and on the degree of abnormality observed. In some cases, e.g. in the application of CS and FCIA to the emergency coordination testbed, we have observed that the H-mechanisms enhance the overall utility of a system with an abnormality even above the case where no abnormality is present (the baseline in our comparisons). This clearly indicates that dynamic modification of adaptation strategies at the architecture level via H-mechanism is viable – just imagine that these potential adaptation strategy modifications were to be defined upfront at a single architectural level.

What is the overhead of using each H-mechanism? The overhead of each H-mechanism depends on the worst-case runtime complexity of each mechanism. For each mechanism, we have provided an upper bound estimate in big-O notation. From this perspective, the mechanism with a higher overhead is the CS H-mechanism, with its complexity being quadratic both in the number of components and the length of knowledge fields considered.

Knowledge field	*Distance metric (μ)*	*Tolerable distance (T)*	*Confidence level (α)*
position	Euclidean	3	0.9
battery	difference	0.005	0.95
dirtinessMap	See Fig. 15	3	0.9

Fig. 14. Distance metrics, tolerable distances, and confidence levels in Robot knowledge fields.

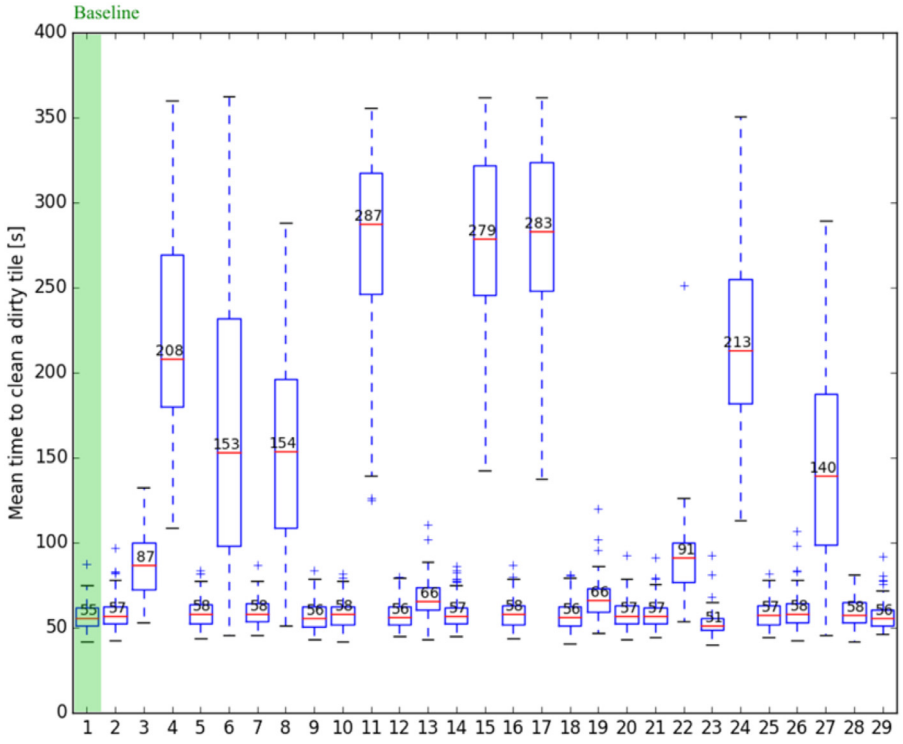


Fig. 16. Utilities of the system when one transition at a time is added in the EMS H-mechanism. Added transitions are detailed in Fig. 17.

Box number	Added transitions
1	Baseline – 4 robots, 1 docking station
2	Battery dead → Charging
3	Cleaning → Blocked
4	Cleaning → Battery dead
5	Battery dead → Cleaning
6	Searching → Charging
7	Approaching dock → Cleaning
8	Blocked → Charging
9	Approaching dock → Searching
10	Battery dead → Approaching dock
11	Charging → Battery dead
12	Searching → Cleaning
13	Charging → Approaching dirt
14	Blocked → Approaching dirt
15	Approaching dirt → Battery dead
16	Blocked → Searching
17	Approaching dirt → Charging
18	Blocked → Cleaning
19	Charging → Cleaning
20	Charging → Approaching dock
21	Battery dead → Approaching dirt
22	Approaching dirt → Blocked
23	Searching → Blocked
24	Cleaning → Charging
25	Battery dead → Searching
26	Approaching dock → Approaching dirt
27	Blocked → Battery dead
28	Charging → Blocked
29	Battery dead → Blocked

Fig. 17. Description for Fig. 16.

Can multiple H-mechanisms be applied at the same time? Depending on the particular fitness function applied, EMS may be triggered in a situation that is also covered by other H-mechanisms (e.g. by CS). Resolving potential conflicts is a task of the H-Adaptation Manager. Here, based on a number of experiments, we have employed the observation that triggers of the P + E phases of the H-mechanisms appear "exceptionally"; therefore we handle the (very special) case that several triggers appear simultaneously in a simplistic way: The H-Adaptation Manager randomly decides which of the triggers will be accepted – the other are ignored and may be re-activated in a future run of the M + A phases of the corresponding H-mechanisms.

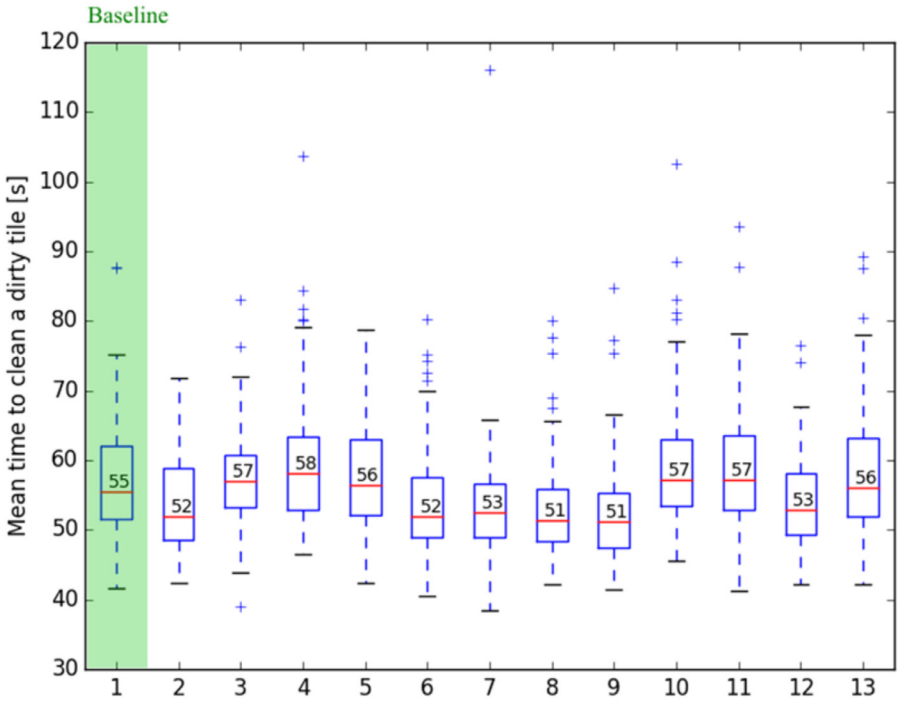


Fig. 18. Utilities of the system when two transitions at a time are added in the EMS H-mechanism. Added transitions are detailed in Fig. 19.

Box number	Added transitions
1	Baseline – 4 robots, 1 docking station
2	Searching → Blocked Approaching dock → Approaching dirt
3	Approaching dock → Approaching dirt Approaching dock → Searching
4	Approaching dock → Searching Battery dead → Blocked
5	Approaching dock → Approaching dirt Battery dead → Blocked
6	Approaching dock → Approaching dirt Searching → Blocked
7	Battery dead → Blocked Searching → Blocked
8	Searching → Blocked Battery dead → Blocked
9	Searching → Blocked Approaching dock → Searching
10	Approaching dock → Searching Approaching dock → Approaching dirt
11	Battery dead → Blocked Approaching dock → Approaching dirt
12	Approaching dock → Searching Searching → Blocked
13	Battery dead → Blocked Approaching dock → Searching

Fig. 19. Description for Fig. 18.

How can new H-mechanisms be developed? The development of an H-mechanism starts by identifying a recurring problem that exists with an adaptation strategy and implementing a solution to the problem that is less intrusive than patching the adaptation strategy itself (which may have other dependencies and thus be difficult to change). An H-mechanism is thus specific to an adaptation strategy.

Generality and Limitations. In terms of generalization of our approach, we have focused on articulating our Assumptions on the Adaptation layer, to make it clear when an H-mechanism can be applied to it. Please note that the specifications of our assumptions are DEECo-independent.

Our approach does not depend on the existence of a centralized entity hosting the homeostasis layer. DEECo itself does not specify any deployment style, i.e. DEECo-based systems can range from completely centralized to completely decentralized ones. In the latter case, the ensemble evaluation is performed independently by each component at run-time. The H-mechanisms we propose in this paper can also be decentralized. The ease and degree of decentralization depends on the H-mechanism: while FCIA can be trivially and completely decentralized, since it relies on component-

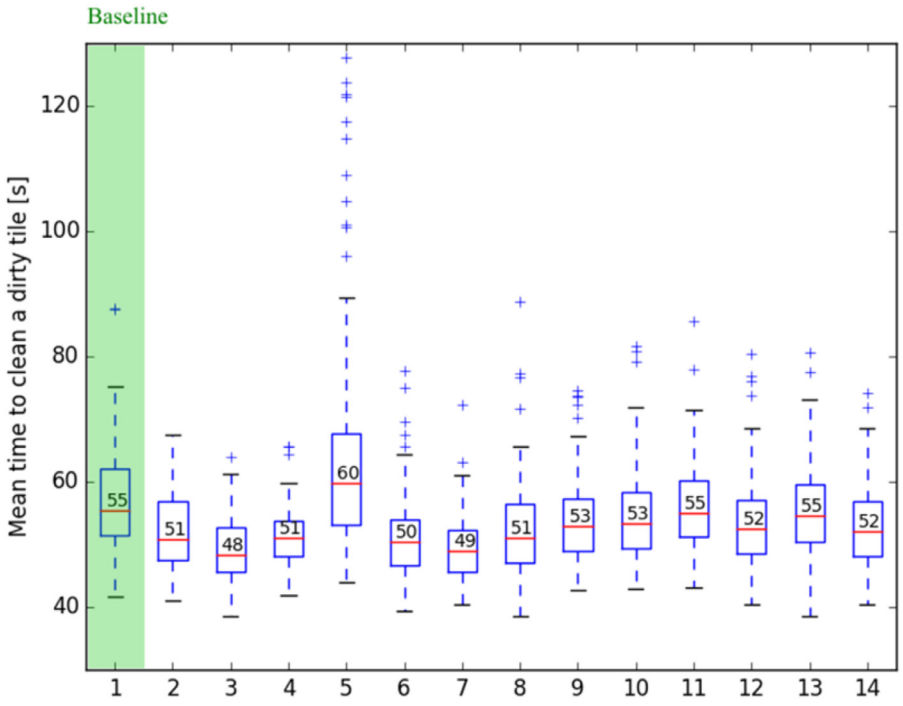


Fig. 20. Utilities of the system when a guard is adjusted using the AG mechanism. Adjusted guards are: charged level – battery level to leave a docking station; drained level – battery level to go to a docking station; found enough – number of found dirt to start cleaning; cleaned enough – number of found dirt to stop cleaning. Guards detailed in Fig. 21.

Box number	Added transitions
1	Baseline – 4 robots, 1 docking station
2	Charged level = 0.9
3	Charged level = 0.7
4	Charged level = 0.5
5	Drained level = 0.1
6	Drained level = 0.3
7	Drained level = 0.5
8	Found enough = 1
9	Found enough = 3
10	Found enough = 7
11	Found enough = 9
12	Cleaned enough = 1
13	Cleaned enough = 2
14	Cleaned enough = 3

Fig. 21. Description for Fig. 20.

local information only, the other three H-mechanisms are difficult to decentralize since they rely on information gathered from a number of components.

Regarding the computational overhead of our approach, we note that, in general, the extra layer demands additional computational load, since monitoring of the triggering events is inherent to all four H-mechanisms. Even though it is minor for CS and FCIA, in the case of EMS and AG it depends on the complexity of the associated fitness function. Obviously, the most computationally demanding step is the data collection in CS if done preventively at run-time. This can be reduced by limiting the time window for collecting data, or by starting it ex-post, i.e. if need be.

Another limitation of the work presented in this paper is that the proposed H mechanisms have been only been evaluated so far with DEECo self-adaptation strategies. Investigating the generalizability of our homeostasis concept with other self-adaptation approaches (e.g. Stitch (Cheng et al., 2012)) is an interesting topic of our future work.

6. Related work

In this paper, we focus on self-adaptive architectures and run-time support for handling run-time uncertainty in the context of software-intensive cyber-physical systems. Therefore, as part of related work, we reflect on research in the areas of architecture-based self-adaptation, addressing uncertainty in self-adaptive systems, and self-adaptation and resilience in cyber-physical systems. This topic touches upon several research works which we overview and compare to our work below. We have grouped related works in three (partially overlapping) research strands.

6.1. Architecture-based self-adaptation

Architecture-based self-adaptation refers to changes in the run-time architecture of a system (e.g. adding/removing components and connectors) to deal with changes in the environment or the goals of a deployed system. The early works of Oreizy et al. (1998) and Kramer and Magee (2007) provide the

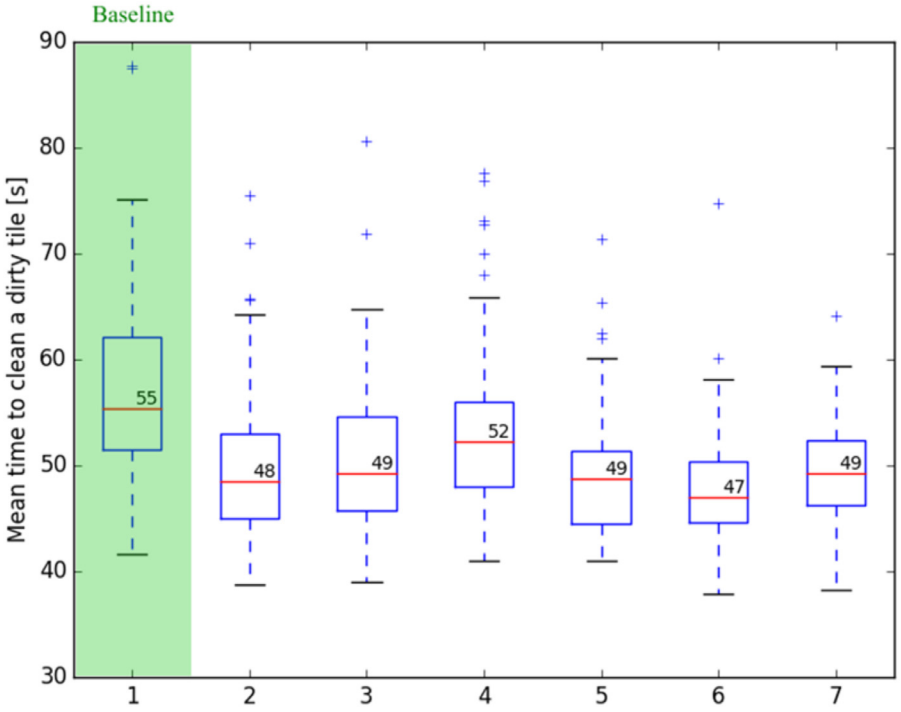


Fig. 22. Utilities of the system when two guards at a time are adjusted in AG mechanism. Adjusted guards: charged level – battery level to leave a docking station; drained level – battery level to go to a docking station; found enough – number of found dirt to start cleaning; cleaned enough – number of found dirt to stop cleaning. Guards detailed in Fig. 23.

Box number	Added transitions
1	Baseline – 4 robots, 1 docking station
2	Drained level = 0.3 Found enough = 3
3	Drained level = 0.3 Cleaned enough = 2
4	Found enough = 3 Cleaned enough = 2
5	Charged level = 0.7 Drained level = 0.3
6	Charged level = 0.7 Found enough = 3
7	Charged level = 0.7 Cleaned enough = 2

Fig. 23. Description for Fig. 22.

foundations of the field—an overview of the literature in this area is provided in Weyns and Ahmad (2013). In the following, we give an overview of the most related approaches that support architecture-based self-adaptation strategies:

An approach representative of architecture-based self-adaptation is the combination of the Rainbow framework (Cheng et al., 2004) with the Stitch language (Cheng et al., 2012). Rainbow and Stich support the reuse of adaptation strategies and infrastructure to apply them. A running system is monitored for violations and appropriate adaptation strategies are employed to resolve them. We have built on these ideas in our previous work on architecture-based self-adaptation using invariant models at run-time—IRM-SA (Gerostathopoulos et al., 2016b). In this work, we propose adaptation strategies that can evolve at run-time in order to support a wider range of scenarios (situations that warrant self-adaptation) while the Rainbow/Stitch and IRM-SA uses pre-defined, non-evolving strategies.

The PLASMA (Tajalli et al., 2010) and Sykes's approach (Sykes et al., 2008) are based on automated reactive planning-as-model-checking. In both approaches, the goal is to synthesize an appropriate sequence of actions on-the-fly which will lead to the satisfaction of a goal. When a goal is not satisfied anymore (e.g. due to some obstacle), automated re-planning is triggered which leads to a new series of actions per component which in turn gets translated in architectural changes (starting/stopping of a component or a component process). Both approaches essentially comply to Kramer and Magee's three-layer reference model for adaptation (Kramer and Magee, 2007) that hoists the concerns of low-level component control, change management, and goal management in different, stacked architecture layers. We, too, decouple the changes of the business system from the changes of the adaptation layer and suggest to handle the former in the adaptation layer, which corresponds to the change management layer in Kramer and Magee's reference model (Kramer and Magee, 2007), and the latter in the homeostasis layer corresponding to the goal management layer in Kramer and Magee's work. Contrary to the above approaches PLASMA and Sykes's approach, which instantiate Kramer and Magee's reference model, we do not synthesize new strategies or plans by invoking a planner, but by changing the existing adaptation strategies using different methods (e.g., data cor-

relation, local search) that are strategy-specific, defined in our H-mechanisms.

MORPH is a reference architecture for the coordination of reconfiguration and behavior adaptation (Braberman et al., 2015). Behavior adaptation refers to a sequence of steps that should be executed while reconfiguration adaptation refers to architecture changes to be executed. MORPH puts forward a Goal Management layer (as in (Sykes et al., 2008)) which re-computes behavior and reconfiguration strategies applied on the two lower layers, the Strategy Management and Enactment to support customizability. While MORPH focuses on increasing the performance of the system for example through parallel architectures, they do not target run-time uncertainty.

Another line of research focuses on the usage of (architectural) models for adaptation. For example, Floch et al. propose to use architectural models for adaptation (Floch et al., 2006) and Sykes et al. propose to revise models through learning for planning in adaptive systems (Sykes et al., 2013). While these works are an important step towards the adaptation based on runtime architectural models and using environmental domain models that are updated at runtime to deal with changing environments, we focus more specifically on dealing with run-time uncertainty in an efficient way.

6.2. Addressing uncertainty in self-adaptive systems

Managing uncertainty has been identified as one of the major challenges in engineering software for self-adaptive systems (Cheng et al., 2009). Self-adaptive systems can be affected by different kinds of uncertainty: Requirements, design and run-time uncertainty (Ramirez et al., 2012a). We reflect here on the major works in addressing uncertainty in self-adaptive systems—for a recent classification of uncertainty in architecture-based self-adaptive systems we refer the interested reader to (Mahdavi-Hezavehi et al., 2016).

On the requirements uncertainty level, Ramirez et al. have introduced the RELAX language which allows to make requirements more tolerant to environmental uncertainty (Ramirez et al., 2012b). In a similar approach, Baresi et al. have proposed FLAGS (Baresi et al., 2010), a modeling method and language for specifying adaptation goals that are fuzzy, i.e. whose satisfaction is the result of a fuzzy logic membership function. Villegas et al. focus on adapting the requirements and context in self-adaptive systems to deal with requirements uncertainty (Villegas et al., 2013). Their DYNAMICO reference model supports dynamic monitoring and requirements variability to allow satisfying system goals under highly changing environments. DYNAMICO supports the adaptation at the model level (i.e., control objectives, context, and context monitors). We focus on supporting self-adaptation at the architectural level.

On design time uncertainty level, Esfahani et al. propose POISED – an approach based on possibility theory for handling internal uncertainty that affects the system in making adaptation decisions (Esfahani et al., 2011). Internal uncertainty is caused by the difficulty of determining the impact of adaptation on the system's quality objectives. Gerostathopoulos et al. proposed the concept of meta-adaptation strategies (inspired from evolutionary-computation approaches to self-adaptation such as AVIDA-MDE (Goldsby and Cheng, 2008), Veritas (Fredericks et al., 2014), and Proteus (Fredericks and Cheng, 2015)) to deal with situations not anticipated at design-time via creating adaptation strategies on the fly and applying them on a target system (Gerostathopoulos et al., 2017). Although our approach also tackles unanticipated situations, we do not specifically rely on evolutionary algorithms to evolve the self-adaptive logic at runtime, but on a number of diverse H-mechanisms. Elkhodary et al. present FUSION that allows a self-adaptive system to self-tune its adaptation logic in case of unanticipated conditions (Elkhodary et al., 2010). It uses a feature-oriented system model and learns the impact of feature selection and feature interaction. In contrast to this, we advocate introducing flexibility in self-adaptation strategies as a method to deal with run-time uncertainty. On run-time uncertainty level, Knauss et al. contribute with ACon – a learning based approach to deal with unpredictable environments and sensor failure (Knauss et al., 2016). It uses machine learning to keep the context in which contextual requirements are valid up-to-date. ACon focus is very similar to the focus of this paper on run-time uncertainty. However, in this paper we take an architectural view and focus on ways to evolve self-adaptive logic at run-time to counteract run-time uncertainty, while ACon focuses on keeping contextual requirements up-to-date. ACon considers uncertainty related to a specific context, while in this work, we take a general view on, for example, tackling sensor failure by adjusting the architecture, instead of dealing with a sensor failure for each requirement individually.

6.3. Self-adaptation and resilience in cyber-physical systems

When looking at self-adaptation in the context of CPS, the most important concerns that warrant adaptation are performance, flexibility, and reliability (Muccini et al., 2016). Since we focus on strengthening a sasiCPS against run-time uncertainty via flexible adaptation strategies, we also position our work against important approaches supporting reliability through resilience (tolerance to faults) at the architecture level.

MetaSelf is a service-oriented architecture framework and accompanying development method for building dynamically resilient systems (Di Marzo Serugendo et al., 2007; Serugendo et al., 2010). The main idea is to encode reconfiguration actions into resilience policies and use metadata encoding functional and non-functional properties to select between services in case a policy is triggered. While they focus on a generic infrastructure for service-oriented dynamically resilient systems, we focus on the flexible evolution of existing adaptation strategies (ensembles, mode-state machines), which broadly correspond to explicitly defined and fixed policies in MetaSelf.

Pradhan et al. focus on resilient operation and execution on the run-time infrastructure level of mobile CPS (Pradhan et al., 2016). Their approach relies on the implicit encoding of all possible mappings of software components to component nodes—configuration points or deployments. When a fault is detected, the infrastructure computes a new configuration point by using information about the failure, the current configuration point, and relevant deployment and resource constraints in the configuration space and invoking an SMT solver. The faults they consider are node (hardware) failures and component (software) failures. In our case, we do not focus on resilience at the infrastructure/deployment level but at the application layer, where more fine-grained analysis of the failure can be achieved. Additionally, we do not rely on a planner but on dynamically changing the existing application-level adaptation strategies in different ways (data correlation, local search) that are strategy-specific (the H-mechanisms).

Important works in resilience management are also furnished in the area of formal methods. For instance, Tarasyuk et al. model a robotic system (similar to our running example) in Event-B (Tarasyuk et al., 2013), a state-based formal development approach relying on iterative refinement of a system's behavioral model. Faults and mitigation strategies are inserted into the model and checked via probabilistic verification (using PRISM). Although we acknowledge the potential of such formal approaches in sasiCPS, we take a different perspective and focus on mitigating run-time uncertainty as we believe it cannot be fully eliminated via model

checking or other verification or testing means in the open and dynamic CPS environment.

7. Conclusions

This paper focused on tackling uncertainty in the operating conditions of self-adaptive software-intensive cyber-physical systems. The general idea is to equip such a system with architecture homeostasis—the ability to change its self-adaptation strategies at run-time according to environment stimuli. This idea was exemplified in four concrete homeostatic mechanisms: Collaborative Sensing, Faulty Component Isolation from Adaptation, Enhancing mode switching, and Adjusting Guards in Mode Switching. Each homeostatic mechanism deals with a particular class of problems related to run-time uncertainty; when triggered, it adjusts its self-adaptation strategies that work at the software architecture level. The conducted experiments have shown that hoisting the modification of self-adaptation strategies at the architectural level is a viable option. We have also tested the feasibility of our approach by applying it in two case studies from different domains.

In our future work, we intend to conduct further research on the classification algorithms to effectively determine situations that trigger homeostatic mechanisms, and investigate, concretize, and experiment with more homeostatic mechanisms.

Acknowledgments

This work was partially supported by the project no. LD15051 from COST CZ (LD) programme by the Ministry of Education, Youth and Sports of the Czech Republic; by Charles University institutional funding SVV-2016-260331 and PRVOUK; by Charles University Grant Agency project No. 391115. This work is part of the TUM Living Lab Connected Mobility project and has been funded by the Bayerisches Staatsministerium fuï Wirtschaft und Medien, Energie und Technologie.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.jss.2018.10.051.

References

Baresi, L., Pasquale, L., Spoletini, P., 2010. Fuzzy goals for requirements-driven adaptation. In: Proc. of RE '10. IEEE, pp. 125–134.
Beetz, K., Böhm, W., 2012. Challenges in engineering for software-intensive embedded systems. In: Pohl, K., Hönninger, H., Achatz, R., Broy, M. (Eds.), Model-Based Engineering of Embedded Systems. Springer, pp. 3–14.
Braberman, V., D'Ippolito, N., Kramer, J., Sykes, D., Uchitel, S., 2015. MORPH: a reference architecture for configuration and behaviour self-adaptation. In: Proceedings of the 1st International Workshop on Control Theory for Software Engineering. ACM, pp. 9–16.
Brun, Y., Serugendo, G.D.M., Gacek, C., Giese, H., Kienle, H., Litoiu, M., Muller, H., Pezze, M., Shaw, M., 2009. Engineering self-adaptive systems through feedback loops. In: Software Engineering for Self-Adaptive Systems. Springer Berlin Heidelberg, pp. 48–70.
Bures, T., Gerostathopoulos, I., Hnetynka, P., Keznikl, J., Kit, M., Plasil, F., 2013. DEECo – an ensemble-based component system. In: Proc. of CBSE'13. ACM, pp. 81–90.
Bures, T., Plasil, F., Kit, M., Tuma, P., Hoch, N., 2016. Software abstractions for component interaction in the internet of things. Computer 49, 50–59.
Cheng, B., de Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Serugendo, G.D.M., Dustdar, S., Finkelstein, A., Cacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H.M., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Muller, H.A., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., Whittle, J., 2009a. Software engineering for self-adaptive systems: a research roadmap. In: Software Engineering for Self-Adaptive Systems. Springer Berlin Heidelberg, pp. 1–26.
Cheng, B., Sawyer, P., Bencomo, N., Whittle, J., 2009b. A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty. In: Proc. of MODELS '09. Springer Berlin Heidelberg, pp. 1–15.
Cheng, S., Huang, A., Garlan, D., Schmerl, B., Steenkiste, P., 2004. Rainbow: Architecture-based self-adaptation with reusable infrastructure. IEEE Comput. 37, 46–54.
Cheng, S.-W., Garlan, D., Schmerl, B., 2012. Stitch: a language for architecture-based self-adaptation. J. Syst. Software 85, 1–38.
Di Marzo Serugendo, G., Fitzgerald, J., Romanovsky, A., Guelfi, N., 2007. A metadata-based architectural model for dynamically resilient systems. In: Proc. of the 2007 ACM Symposium on Applied Computing - SAC '07. ACM Press 566–566.
Elkhodary, A., Esfahani, N., Malek, S., 2010. FUSION: a framework for engineering self-tuning self-adaptive software systems. In: Proc. of FSE '10. ACM, pp. 7–16.
Escoffier, C., Hall, R.S., Lalanda, P., 2007. iPOJO: an extensible service-oriented component framework. In: IEEE International Conference on Services Computing (SCC 2007), pp. 474–481.
Esfahani, N., Kouroshfar, E., Malek, S., 2011. Taming uncertainty in self-adaptive software. In: Proc. of SIGSOFT/FSE '11. ACM, pp. 234–244.
Fairbanks, G., 2014. Architectural hoisting, IEEE Software, 31.
Floch, J., Hallsteinsen, S., Stav, E., Eliassen, F., Lund, K., Gjorven, E., 2006. Using architecture models for runtime adaptability. IEEE Software 23, 62–70.
Fredericks, E.M., Cheng, B.H.C., 2015. Automated generation of adaptive test plans for self-adaptive systems. In: Proc. of SEAMS '15. IEEE, pp. 157–168.
Fredericks, E.M., DeVries, B., Cheng, B.H.C., 2014. Towards run-time adaptation of test cases for self-adaptive systems in the face of uncertainty. In: Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems - SEAMS 2014. ACM Press, pp. 17–26.
Gerostathopoulos, I., Bures, T., Hnetynka, P., Hujecek, A., Plasil, F., Skoda, D., 2017. Strengthening adaptation in cyber-physical systems via meta-adaptation strategies. ACM Trans. Cyber Phys. Syst. 1, 13:1–13:25.
Gerostathopoulos, I., Bures, T., Hnetynka, P., Keznikl, J., Kit, M., Plasil, F., Plouzeau, N., 2016b. Self-adaptation in software-intensive cyber–physical systems: from system goals to architecture configurations. J. Syst. Software 122, 378–397.
Gerostathopoulos, I., Skoda, D., Plasil, F., Bures, T., Knauss, A., 2016a. Architectural homeostasis in self-adaptive software-intensive cyber-physical systems. In: Proceedings of ECSA 2016. Springer, Copenhagen, Denmark, pp. 113–128.
Goldsby, H., Cheng, B., 2008. Automatically generating behavioral models of adaptive systems to address uncertainty. In: Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems, MoDELS'08. Springer Berlin Heidelberg, pp. 568–583.
Hall, R., Pauls, K., McCulloch, S., Savage, D., 2011. OSGi in Action: Creating Modular Applications in Java. Manning Publications, Stamford, CT.
Hoelzl, M., Rauschmayer, A., Wirsing, M., 2008. Engineering of software-intensive systems: state of the art and research challenges. In: Software-Intensive Systems and New Computing Paradigms, pp. 1–44.
Iftikhar, M.U., Weyns, D., 2014. ActivFORMS: active formal models for self-adaptation. In: SEAMS '14. ACM Press, pp. 125–134.
Kephart, J., Chess, D., 2003. The vision of autonomic computing. Computer 36, 41–50.
Kim, B.K., Kumar, P.R., 2012. Cyber–physical systems: a Perspective at the Centennial. Proc. IEEE. 100, 1287–1308.
Knauss, A., Damian, D., Franch, X., Rook, A., Müller, H.A., Thomo, A., 2016. ACon: A learning-based approach to deal with uncertainty in contextual requirements at runtime. Inf. Softw. Technol. 70, 85–99.
Kramer, J., Magee, J., 2007. Self-managed systems: an architectural challenge. In: Proc. of FOSE'07. IEEE, pp. 259–268.
Mahdavi-Hezavehi, S., Avgeriou, P., Weyns, D., 2016. A classification framework of uncertainty in architecture-based self-adaptive systems with multiple quality requirements. In: Managing Trade-offs in Adaptable Software Architectures. Elsevier, pp. 45–78.
Muccini, H., Sharaf, M., Weyns, D., 2016. Self-adaptation for Cyber-physical Systems: A Systematic Literature Review. In: Proceedings of the 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. ACM, New York, NY, USA, pp. 75–81.
Oreizy, P., Medvidovic, N., Taylor, R.N., 1998. Architecture-based runtime software evolution. In: Proc. of ICSE '98. IEEE, pp. 177–186.
Perrouin, G., Morin, B., Chauvel, F., Fleurey, F., Klein, J., Traon, Y.L., Barais, O., Jezequel, J.-M., 2012. Towards flexible evolution of dynamically adaptive systems. In: Proc. of ICSE '12. IEEE, pp. 1353–1356.
Pradhan, S., Dubey, A., Levendovszky, T., Kumar, P.S., Emfinger, W.A., Balasubramanian, D., Otte, W., Karsai, G., 2016. Achieving resilience in distributed software systems via self-reconfiguration. J. Syst. Softw. 122, 344–363.
Ramirez, A.J., Cheng, B.H.C., Bencomo, N., Sawyer, P., 2012b. Relaxing claims: coping with uncertainty while evaluating assumptions at run time. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (Eds.), Model Driven Engineering Languages and Systems. Springer Berlin Heidelberg, pp. 53–69.
Ramirez, A.J., Jensen, A.C., Cheng, B.H., 2012a. A taxonomy of uncertainty for dynamically adaptive systems. In: SEAMS '12. IEEE, pp. 99–108.
Serugendo, G.D.M., Fitzgerald, J., Romanovsky, A., 2010. MetaSelf – an architecture and a development method for dependable self- * systems. In: Proc. of the 2010 ACM symposium on Applied computing - SAC '10. ACM, pp. 457–461.
Shaw, M., 2002. "Self-healing": softening precision to avoid brittleness: position paper for woss '02: workshop on self-healing systems. In: Proceedings of the First Workshop on Self-healing Systems. ACM, pp. 111–114.
Snyman, Jan, 2005. Practical Mathematical Optimization. Springer Science & Business Media.
Srinivasan, S., Mycroft, A., 2008. Kilim: isolation-typed actors for Java. In: ECOOP 2008 – Object-Oriented Programming. Springer, Berlin, Heidelberg, pp. 104–128.
Sykes, D., Corapi, D., Magee, J., Kramer, J., Russo, A., Inoue, K., 2013. Learning revised models for planning in adaptive systems. In: Proceedings of the 2013 International Conference On Software Engineering. IEEE Press, Piscataway, NJ, USA, pp. 63–71.

Sykes, D., Heaven, W., Magee, J., Kramer, J., 2008. From goals to components: a combined approach to self-management. In: Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems (SEAMS '08), pp. 1–8.

Tajalli, H., Garcia, J., Edwards, G., Medvidovic, N., 2010. PLASMA: a plan-based layered architecture for software model-driven adaptation. In: Proceedings of the IEEE/ACM international conference on Automated software engineering. ACM, pp. 467–476.

Tarasyuk, A., Pereverzeva, I., Troubitsyna, E., Laibinis, L., 2013. Formal development and quantitative assessment of a resilient multi-robotic system. In: International Workshop on Software Engineering for Resilient Systems. Springer, pp. 109–124.

Villegas, N.M., Tamura, G., Müller, H.A., Duchien, L., Casallas, R., 2013. DYNAMICO: A reference model for governing control objectives and context relevance in self-adaptive software systems. In: Lemos, R.de, Giese, H., Müller, H.A., Shaw, M. (Eds.), Software Engineering for Self-Adaptive Systems II. Springer Berlin Heidelberg, pp. 265–293.

Weyns, D., Ahmad, T., 2013. Claims and evidence for architecture-based self-adaptation: a systematic literature review. In: Drira, K. (Ed.), Software Architecture. Springer Berlin Heidelberg, pp. 249–265.

Weyns, D., Malek, S., Andersson, J., 2010. FORMS: A formal reference model for self-adaptation. In: Proceedings of the 7th International Conference on Autonomic Computing. ACM, New York, NY, USA, pp. 205–214.

Wolfinger, R, 2012. Dynamic application composition with plux .NET. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.208.7307&rep=rep1&type=pdf.