Architectural Homeostasis in Self-Adaptive Software-Intensive Cyber-Physical Systems

Ilias Gerostathopoulos[1,2], Dominik Skoda[2], Frantisek Plasil[2], Tomas Bures[2], Alessia Knauss[3]

[1]Fakultät fur Informatik, Technische Universität München.
Munich, Germany
[2]Charles University in Prague, Faculty of Mathematics and Physics.
Prague, Czech Republic
[3]Department of Computer Science and Engineering, Chalmers University of Technology.
Gothenburg, Sweden

gerostat@in.tum.de
{skoda,plasil,bures}@d3s.mff.cuni.cz
alessia.knauss@chalmers.se

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 describe three mechanisms that reify the idea: (i) collaborative sensing, (ii) faulty component isolation from adaptation, and (iii) enhancing mode switching. Moreover, our experimental evaluation of the three mechanisms confirms that allowing a complex system to change its self-adaptation strategies helps the system recover from runtime errors and abnormalities and keep it in an operational state.

Keywords: cyber-physical systems; software architecture; run-time uncertainty; self-adaptation strategies

1 Introduction

Cyber-Physical Systems (CPS) [1] are large complex systems that rely more and more on software for their operation—they are becoming *software-intensive* CPS [2, 3]. Such systems, e.g., intelligent transportation systems, smart grids, 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 properties in order to actuate physical processes. Due to the close connection to the physical world that is hard to predict at

design time and control at run-time, they encounter a high level of uncertainty in their operating conditions—*run-time uncertainty* [4]. Such kind of uncertainty 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 complete system crash [4].

A promising way to tackle run-time uncertainty is to endow software-intensive CPS with self-adaptive capabilities, i.e., with capabilities of adjusting their own structure and behavior at run-time based on their internal state and the perceived environment, while considering their run-time goals and requirements [5]. In our work, we focus on self-adaptation approaches implemented at the architectural level (e.g. Stitch [6], [7–10]). 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 [11]. 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 [4, 12].

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 [13], by specifically focusing at 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. The last point is common in other domains (e.g., communication protocols that try to reestablish a connection in some random manner in order to avoid a flood of reconnections).

The main contribution of this paper lies in presenting three concrete homeostatic 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 mechanisms in a development and run-time framework for sasiCPS—*DEECo* component framework [14]—and in evaluating their feasibility and effectiveness in a controlled experiment. In the experimental setup, the mechanisms worked both independently and in combination with each other. The results show that using the proposed mechanisms increase in the overall utility of the system in face of runtime errors and abnormalities (high-level exceptions).

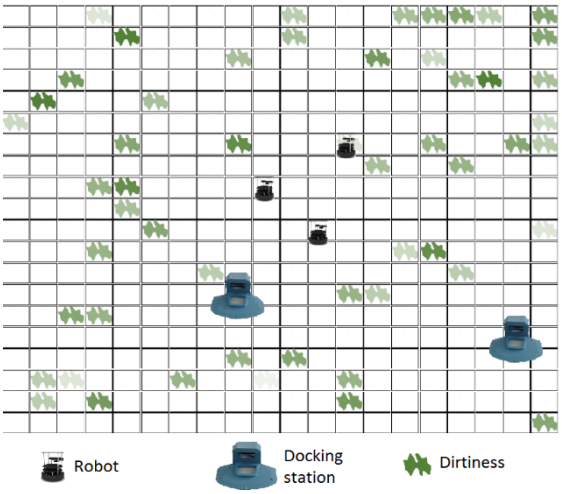

Fig. 1. Cleaning robots example (screenshot from the tool).

The rest of the paper is structured as follows. Section 2 presents our running example and the background of our work. Section 3 presents the main idea of architectural homeostasis, together with its reification into three concrete homeostatic mechanisms. Section 4 details our evaluation based on implementing the mechanisms and quantifying their effects in the running example, together with discussing the interesting points, extensions, and limitation of our approach. Finally, Section 5 compares our work to existing ones in the literature and Section 6 concludes.

2 Running Example and Background

2.1 Cleaning Robots Example

In the scenario 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 three robots and two docking stations.

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 a number of situations where run-time uncertainty creeps in. These include situations where a robot loses the ability of reliably detecting dirty tiles (e.g. due to a failure in its humidity sensor) or loses the ability of communicating with the docking stations. Docking stations may also stop working.

```
1.    role Dockable:
2.      id, assignedDockingStationPosition
3.    role Cleaner:
4.      id, position, targetPosition, dirtinessMap
5.    role Dock:
6.      position, dockedRobots
7.
8.  component Robot1 features Dockable, Cleaner
9.      knowledge:
10.       id = 2
11.       position = { 16, 02}
12.       dirtinessMap = {}
13.       targetPosition = null
14.       assignedDockingStationsPosition = null
15.
16.     process move in mode Cleaning, Searching
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 Robot1 */
34.    /* similar spec for Robot2 and Robot3, DockingStation1, and DockingStation2 */
```

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

Run-time uncertainty is also manifested in the unpredictable pace and position where dirt appears in the space.

2.2 DEECo Model of Cleaning Robots – Running Example

DEECo is a development and run-time framework for sasiCPS [14]. In DEECo, a component is an independent entity of development and deployment. Two component types were identified in our running example: Robot and DockingStation. 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). In the running example, each Robot 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

```
35.  ensemble DockingInformationExchange:
36.   coordinator: Dock
37.   member: Dockable
38.  membership:
39.   coordinator.dockedRobots.size() <= 3
40.  knowledge exchange:
41.   coordinator.dockedRobots ← member.id
42.   member.assignedDockingStationPosition  ← coordinator.position
43.  scheduling: periodic( 1000ms )
44.
45.  ensemble CleaningPlanExclusion:
46.    coordinator: Cleaner
47.    member: Cleaner
48.   membership:
49.    coordinator.targetPosition == member.targetPosition
50.      and distance(coordinator.position, coordinator.targetPosition)
51.           < distance(member.position, member.targetPosition)
52.   knowledge exchange:
53.    member.targetPosition ← null
54.  scheduling: periodic( 1000ms )
```

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

machine (Fig. 7). 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. An ensemble is dynamically created/disbanded depending on which components satisfy its membership condition. The key task of an ensemble is to periodically exchange knowledge parts between its coordinator and member components (determined by their roles). At design-time, an ensemble specification consists of (i) ensemble roles that the member and coordinator components should feature, and (ii) a membership condition prescribing the condition under which components should interact (Fig. 3, lines 45-51), and (iii) a knowledge exchange function, which specifies the knowledge exchange that takes place between the components in the ensemble (lines 52-53). For instance, the components featuring the Dockable role (e.g. Robot) can form an ensemble with components featuring the Dock role (i.e. with a DockingStation) to coordinate on the docking activity (lines 36-37).

Matching of 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. This semantics provides for software architecture that is dynamically adapted to the current components' knowledge values.

Self-Adaptation in DEECo. The semantics of switching modes within a component reflects the idea of the MAPE-K self-adaptation loop (Monitor-Analyze-Plan-Execute over Knowledge) [15]: 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

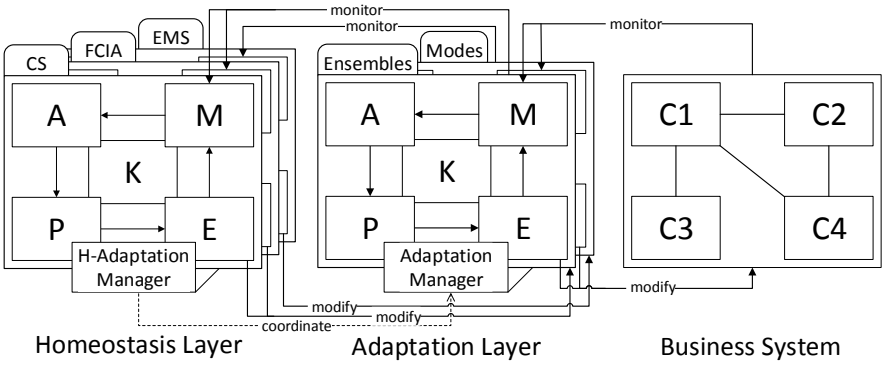

Fig. 4. Three-layered architecture with homeostasis layer

analyzed which of the eligible transition should be selected by so that the next mode is planned. Finally, the next mode is brought to action (executed).

The semantics of ensembles also reflects the idea of MAPE-K: 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 of these 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 then realized.

Overall, 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 a self-adaptation mechanism defines a particular self-adaptation strategy (in the sense of [6]), 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 runtime framework of DEECo [14]), 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 environment assumptions 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. [16]. 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 self-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 three H-mechanisms. H-Adaptation manager coordinates monitoring of exceptional/unanticipated situations at the Adaptation Layer and reacts by activation of a selected H-mechanism, which, in turn, modifies a self-adaptation strategy at the Adaptation Layer. Technically, the Adaptation Manager coordinates the application of self-adaptation mechanisms (to avoid conflicts in adaptation); a similar coordination role has the H-Adaptation Manager with respect to application of H-mechanisms. Moreover, the H-Adaptation manager can force the Adaptation Manager to postpone any adaptation based on the self-adaptation strategy being modified.

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 the 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 (mode-state machines, ensembles), by modifying/adding/removing the self-adaptation strategies defined by their instances at run-time.

3.1 H-Mechanism #1: Collaborative Sensing

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 for satisfying important functional requirements (e.g. a robot needs to know its position in order to plan its path to a destination), it becomes extremely important to deal with sensor malfunctioning to still enable environment sensing at run-time.

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 similar property P. Collaborative sensing (CS) H-mechanism provides an adequate approximation of property P 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 computational steps: (i) *CS Analysis*—identification of data dependencies and (ii) *CS Plan*—approximation of P. While CS Plan is relatively easy to realize once a dependency relation is identified, for the two main tasks of CS Analysis (Fig. 5), there are multiple alternatives involving different trade-offs: A major issue is the computational overhead of data collection vs. the readiness of dependency relation when the need for applying CS is acute.

For illustration of CS Analysis, in the following we consider the subtasks (a)-(i)-(I) and (b)-(i). Let us assume that the H-Adaptation Manager 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

a. Data collection
- (i) On a real system
  - I. Preventively
  - II. Ex-post
- (ii) By simulation

b. Acquiring dependency relation
- (i) Regression/machine learning
- (ii) Empirical knowledge

Fig. 5. Alternative Subtasks of CS Analysis (identification of data dependencies).

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 Analyze computes the boundary Δ_{k_l} 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 CS Execute.

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.

For illustration, consider the situation where the downwards-looking camera of a robot R starts failing and consequently R loses the ability to detect dirtiness on the floor (and, to update its dirtinessMap). This situation will trigger 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. 6) and can resume its cleaning operation.

```
55. role DirtinessMapRole:
56.   position, dirtinessMap
57.
58. ensemble DirtinessMapExchange:
59.   coordinator: DirtinessMapRole
60.   member: DirtinessMapRole
61.  membership:
62.   // Member and coordinator must be "close" to form the ensemble
63.   // The robot with broken sensor becames the coordinator
64.   close(coordinator.position, member.position)
65.     and obsolete(coordinator.dirtinessMap)
66.  knowledge exchange:
67.       coordinator.dirtinessMap ← member.dirtinessMap
68.  scheduling: periodic( 1000ms )
```

Fig. 6. DSL excerpt from specification of collaborative sensing ensemble of the cleaning robots.

3.2 H-Mechanism #2: Faulty Component Isolation from Adaptation

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 *P*. In such a case, FCIA modifies the adaptation strategies that count on *P* in order to prevent the "contamination" of other components with faulty values of *P*.

For illustration, consider a situation where a docking station *DS* due to some error or malfunction is not able of having docked robots charge anymore, while still being advertised as operational to robots, which 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 FCIA H-mechanism by the H-Adaptation Manager. 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*.

3.3 H-Mechanism #3: Enhancing Mode Switching

The motivation behind the enhancing mode switching (EMS) H-mechanism is that there are cases where the behavior of a component specified by its mode-state machine is over-constrained. 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. Specifically, EMS (i) creates new probabilistic transitions from every mode to every other one, and (ii) introduces probabilities to the existing mode transitions (in Fig. 7 the introduced probabilities have a value of 0.01; however, the actual probability learned in our experiments is much smaller). The

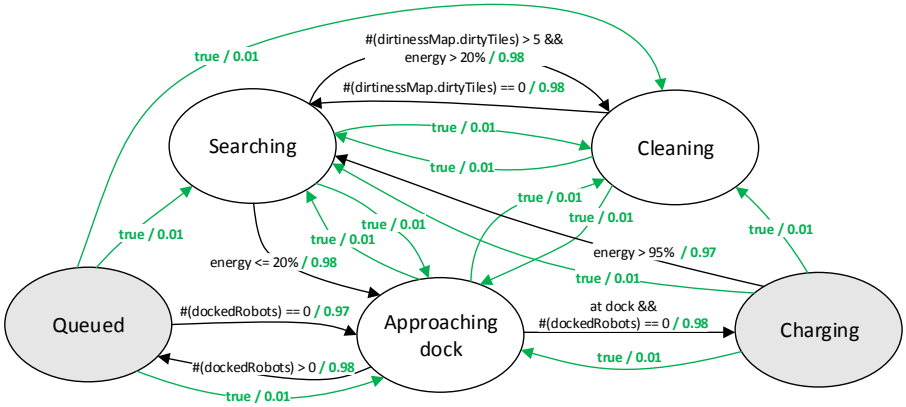

Fig. 7. Mode-state machine capturing the mode switching logic of the Robot component. Each state (mode) is associated with several processes. Transitions are guarded by conditions upon the Robot's knowledge. Changes introduced by the EMS H-mechanism are marked in (bold) green – transitions are now guarded by a condition/probability pair. States that are not allowed to have incoming transitions are marked in grey background.

resulting mode-state machine is represented by a fully connected probabilistic graph. An important part of EMS is a fitness function assessing the impact of a specific modification to the mode-state machine (e.g. by evaluation of system performance). EMS monitors the value of the fitness function and triggers the change of probabilities when the value is low. This change is subject to iterations to tune the fitness value to the desired threshold (e.g. by simulated annealing).

For illustration, consider the unanticipated situation that 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. 7, they might all switch to Charging mode at similar points in time (when their energy falls below 20%). This would result in an increase in the average charging time, since robots will need to queue up at the docking stations. The situation when the queuing and consequently charging time of robots takes longer than usual will act as a trigger for EMS. It will change the mode-state machine of affected robots by adding new transitions and guards (depicted in green in Fig. 7). The new transitions have a probability of 0.01. This is just for illustration; the actual probabilities learned in our experiments are much smaller (see Section 4.2). It is important to realize that each component may find itself in the triggering situation of EMS at a different time and that the mode-state machine evolution is also specific to an individual component.

The EMS H-mechanism effectively allows the transition from every mode to every other mode with a given probability. This, however, can be dangerous when there exist modes that should be entered only under certain circumstances (e.g., because they involve operations with non-revertible effects). To address this issue, we assume that there is a way to specify such forbidden transitions in the mode-state machine.

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

Fig. 8. Scenarios considered in the controlled experiment. Simulation duration is 600 s (with extra 300 secs "learning phase" in scenarios 7 & 9), environment size is 20 x 20, number of robots is 4.

4 Evaluation and Discussion

4.1 Experiment Design and Testbed

In order to quantitatively assess the effects of the three H-mechanisms, we applied them to the running example (Section 2.1) in its JDEECo implementation (JDEECo is a Java implementation of the DEECo component model [14]). We implemented them as plugins to the JDEECo framework, taking advantage of its modeling and simulation capabilities. All these realizations of H-mechanisms[1] are governed by the H-Adaptation Manager implemented as an isolated DEECo component.

To show that the application of the H-mechanisms increases the overall utility of the system in case of faults, a controlled experiment was designed and conducted. This was based on a number of simulation runs of predefined scenarios, each being a combination of deliberatively introduced faults to be addressed by a particular H-mechanism (or their combination). In total, we considered the 8 different scenarios, each of them containing four robots, depicted in Fig. 8. To measure the overall utility of a system run, we used an application-specific metric returning the 90th percentile of the time required for a tile that got dirty until it is cleaned.

4.2 Results and Discussion

Results. Each scenario was run in 100 iterations. Fig. 9 shows the values of the overall system utility in the form of boxplot diagrams where the number associated with the red line denotes the median of the sample. System utility is expressed by the time needed to clean a tile after it gets dirty, the smaller the time the better.

Scenario 1 represents the vanilla case (no faults – no H-mechanism active), acting as the baseline. Not surprisingly, in other scenarios the 90th percentile of the time to clean a tile increases when a fault occurs and is not counteracted by an H-mechanism (scenarios 2,4,6,8). When an H-mechanism counteracts the fault (scenarios 3,5,7,9), the

[1] Available at: https://github.com/d3scomp/uncertain-architectures

overall utility improves, but does not reach the baseline scenario. Below we comment more on CS and EMS, since the application of FCIA was straightforward.

As to the application of CS (scenario 2), a dependency relation (Section 3.1) was identified such that the closeness of the positions of Robot components implied similar values in their dirtinessMaps. This resulted in the creation and deployment of the DirtinessMapExchange of Fig. 6. The used metrics, tolerable distances, and confidence levels are depicted in Fig. 10.

The effect of EMS is illustrated in scenarios 6 and 7. In both scenarios, only a single docking station is active, corresponding to the situation that one of the two docking stations gets unavailable at run-time. When EMS is applied (scenario 7), due to the introduced probabilistic mode switching, the robots started visiting the docking station at different times. Hence, the overall queueing time was reduced and the overall utility increased. EMS needs time to auto-calibrate (set to 300 sec) as it searches for the probability value for the added transitions that yields the highest fitness value following a simulated annealing algorithm. In Fig. 9, the results have been split into the learning phase (7a) and the execution run with learned values (7b). The solution naturally underperforms in the learning phase compared to the case without EMS (6) because of the trial and error that the learning involves. However, once the learning period is over and EMS uses the learned values, it yields a significantly better behavior compared to (6). The fitness value was calculated as the inverse of the average time it takes the robot to clean a tile since it discovered the dirt. Since EMS was running independently for each robot, the local searches returned different optimal probabilities for each robot found by the search (with values close to 0.0001).

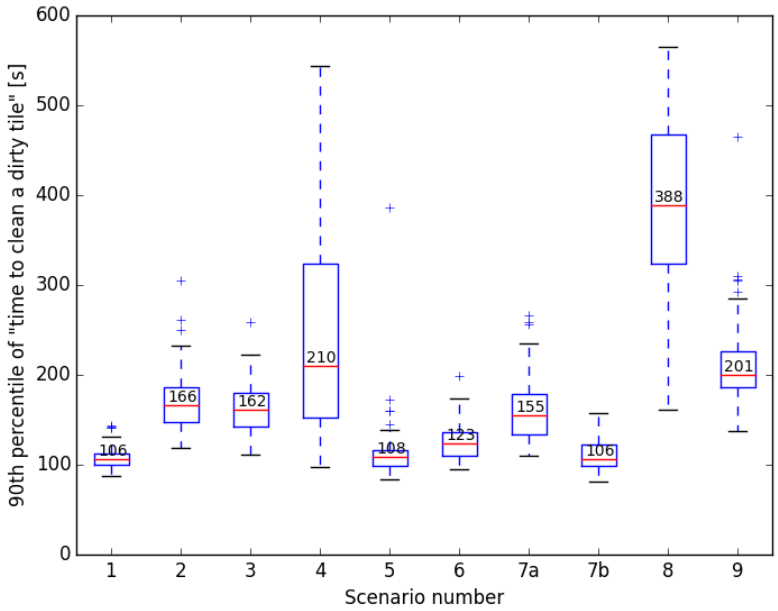

Fig. 9. Simulation results. Smaller values are better.

Knowledge field	*Distance metric (μ)*	*Tolerable distance (T)*	*Confidence level (a)*
position	Euclidean	3	0.9
battery	difference	0.005	0.95
dirtinessMap	In Fig. 11	3	0.9

Fig. 10. Distance metrics, tolerable distances, and confidence levels in Robot knowledge fields.

```
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. 11. Distance metric for dirtinessMap field of Robot component.

In scenario (8) all the faults are introduced and in (9) they are handled by all the three H-mechanisms; this illustrates that all of them can be active at the same time without worsening the overall utility of the system. Since the fitness function in EMS was selected in such a way that it does not depend on the faults triggering CS and FCIA, all the three H-mechanisms behaved as orthogonal.

Discussion. We use two distinct architectural layers—"standard" self-adaptation and adaptation of self-adaptation strategies (the task of H-mechanisms). Hence, our solution basically follows the principle of architectural hoisting [17]—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.

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). In such a case it is important to address this interference and state which H-mechanism has precedence in order to avoid unnecessary side effects. This is the task of the H-Adaptation Manager.

Limitations. In general, the extra layer demands additional computational load, since monitoring of the triggering events is inherent to all three H-mechanisms. Even though it is minor for CS and FCIA, in the case of EMS 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 runtime. This can be reduced by limiting the time window for collecting data, or by starting it ex-post, i.e. when a need be.

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

5 Related Work

In this paper we focus on handling run-time uncertainty in the context of sasiCPS engineered as self-adaptive systems. We thus discuss related literature on the topics of handling uncertainty in cyber-physical as well as self-adaptive systems, in addition to works on solving run-time architecture problems.

Managing uncertainty has been identified as one of the major challenges in engineering software for self-adaptive systems [5]. Self-adaptive systems can be affected by different kinds of uncertainty: Requirements, design and run-time uncertainty [4]. We reflect on the major works in uncertainty affecting self-adaptive systems. On the requirements uncertainty level, Ramirez et al. have introduced the RELAX language which allows to make requirements more tolerant to environmental uncertainty [18]. Esfahani et al. propose POISED – an approach based on possibility theory for handling internal uncertainty that affects the system in making adaptation decisions [19]. Internal uncertainty is caused by the difficulty of determining the impact of adaptation on the system's quality objectives. Knauss et al. contribute with ACON – a learning based approach to deal with unpredictable environment and sensor failure [20]. It uses machine learning to keep the context in which contextual requirements are valid up-to-date. In contrast to the approaches discussed on handling uncertainty in self-adaptive systems, only ACON focuses on the same kind of uncertainty as we do in this paper – the 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 requirements up to date.

On architecture-based run-time adaptation, the works by Oreizy et al. [21] on the adaptation and evolution management and Garlan et al. on the Rainbow framework [22] are important. Rainbow supports 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. However, only predesigned strategies are used, which also do not evolve at run-time.

Elkhodary et al. present FUSION that allows a self-adaptive system to self-tune its adaptation logic in case of unanticipated conditions [23]. It uses a feature-oriented system model and learns the impact of feature selection and feature interaction. In contrast to this, we do not use a learning-based approach, but advocate introducing flexibility in self-adaptation strategies as a method to deal with run-time uncertainty. Villegas et al. focus on supporting context-awareness in self-adaptive systems [24]. Their DYNAMICO reference model supports dynamic monitoring and requirements variability to allow satisfying system goals under highly changing environments.

DYNAMICO supports adaptation at the model level (i.e., control objectives, context, and context monitors). We focus on supporting self-adaptation at the architectural level.

6 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 three concrete homeostatic mechanisms, which, when triggered, adjust self-adaptation strategies that work at the software architecture level. The conducted experiments showed that hoisting modification of self-adaptation strategies at the architectural level is a viable option.

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.

Acknowledgements. 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 fundings 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 für Wirtschaft und Medien, Energie und Technologie.

References

1. Kim, B.K., Kumar, P.R.: Cyber–Physical Systems: A Perspective at the Centennial. Proc. IEEE. 100, 1287–1308 (2012).
2. Hoelzl, M., Rauschmayer, A., Wirsing, M.: Engineering of Software-Intensive Systems: State of the Art and Research Challenges. In: Software-Intensive Systems and New Computing Paradigms. pp. 1–44 (2008).
3. Beetz, K., Böhm, W.: Challenges in Engineering for Software-Intensive Embedded Systems. In: Pohl, K., Hönninger, H., Achatz, R., and Broy, M. (eds.) Model-Based Engineering of Embedded Systems. pp. 3–14. Springer (2012).
4. Ramirez, A.J., Jensen, A.C., Cheng, B.H.: A taxonomy of uncertainty for dynamically adaptive systems. In: SEAMS '12. pp. 99–108. IEEE (2012).
5. Cheng, B., et al.: Software Engineering for Self-Adaptive Systems: A Research Roadmap. In: Software Engineering for Self-Adaptive Systems. pp. 1–26. Springer Berlin Heidelberg. (2009).
6. Cheng, S.-W., Garlan, D., Schmerl, B.: Stitch: A language for architecture-based self-adaptation. J. Syst. Softw. 85, 1–38 (2012).
7. Iftikhar, M.U., Weyns, D.: ActivFORMS: Active Formal Models for Self-Adaptation. In: SEAMS '14. pp. 125–134. ACM Press (2014).
8. Weyns, D., Malek, S., Andersson, J.: FORMS: A Formal Reference Model for Self-adaptation. In: Proceedings of the 7th International Conference on Autonomic Computing. pp. 205–214. ACM, New York, NY, USA (2010).

9. Floch, J., Hallsteinsen, S., Stav, E., Eliassen, F., Lund, K., Gjorven, E.: Using architecture models for runtime adaptability. IEEE Softw. 23, 62–70 (2006).
10. Brun, Y., Serugendo, G.D.M., Gacek, C., Giese, H., Kienle, H., Litoiu, M., Muller, H., Pezze, M., Shaw, M.: Engineering Self-Adaptive Systems through Feedback Loops. In: Software Engineering for Self-Adaptive Systems. pp. 48–70. Springer Berlin Heidelberg (2009).
11. Gerostathopoulos, I., Bures, T., Hnetynka, P., Hujecek, A., Plasil, F., Skoda, D.: Meta-Adaptation Strategies for Adaptation in Cyber-Physical Systems. In: Proc. of ECSA '15. pp. 45–52. Springer (2015).
12. Cheng, B., Sawyer, P., Bencomo, N., Whittle, J.: A Goal-Based Modeling Approach to Develop Requirements of an Adaptive System with Environmental Uncertainty. In: Proc. of MODELS '09. pp. 1–15. Springer Berlin Heidelberg (2009).
13. Shaw, M.: "Self-healing": Softening Precision to Avoid Brittleness. In: Proceedings of the First Workshop on Self-healing Systems. pp. 111–114. ACM (2002).
14. Bures, T., Gerostathopoulos, I., Hnetynka, P., Keznikl, J., Kit, M., Plasil, F.: DEECo – an Ensemble-Based Component System. In: Proc. of CBSE'13. pp. 81–90. ACM (2013).
15. Kephart, J., Chess, D.: The Vision of Autonomic Computing. Computer. 36, 41–50 (2003).
16. Perrouin, G., Morin, B., Chauvel, F., Fleurey, F., Klein, J., Traon, Y.L., Barais, O., Jezequel, J.-M.: Towards Flexible Evolution of Dynamically Adaptive Systems. In: Proc. of ICSE '12. pp. 1353–1356. IEEE (2012).
17. Fairbanks, G.: Architectural Hoisting. IEEE Softw. 31, (2014).
18. Ramirez, A.J., Cheng, B.H.C., Bencomo, N., Sawyer, P.: Relaxing Claims: Coping with Uncertainty While Evaluating Assumptions at Run Time. In: Model Driven Engineering Languages and Systems. pp. 53–69. Springer, (2012).
19. Esfahani, N., Kouroshfar, E., Malek, S.: Taming uncertainty in self-adaptive software. In: Proc. of SIGSOFT/FSE '11. pp. 234–244. ACM (2011).
20. Knauss, A., Damian, D., Franch, X., Rook, A., Müller, H.A., Thomo, A.: ACon: A learning-based approach to deal with uncertainty in contextual requirements at runtime. Inf. Softw. Technol. 70, 85–99 (2016).
21. Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-based Runtime Software Evolution. In: Proc. of ICSE '98. pp. 177–186. IEEE (1998).
22. Cheng, S., Huang, A., Garlan, D., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. IEEE Comput. 37, 46–54 (2004).
23. Elkhodary, A., Esfahani, N., Malek, S.: FUSION: A Framework for Engineering Self-tuning Self-adaptive Software Systems. In: Proc. of FSE '10. pp. 7–16. ACM (2010).
24. Villegas, N.M., Tamura, G., Müller, H.A., Duchien, L., Casallas, R.: DYNAMICO: A Reference Model for Governing Control Objectives and Context Relevance in Self-Adaptive Software Systems. In: Software Engineering for Self-Adaptive Systems II. pp. 265–293. Springer Berlin Heidelberg (2013).