Strengthening Adaptation in Cyber-Physical Systems via Meta-Adaptation Strategies

ILIAS GEROSTATHOPOULOS, Technische Universität München
TOMAS BURES, Charles University in Prague
PETR HNETYNKA, Charles University in Prague
ADAM HUJECEK, Charles University in Prague
FRANTISEK PLASIL, Charles University in Prague
DOMINIK SKODA, Charles University in Prague

The dynamic nature of complex Cyber-Physical Systems puts extra requirements on their functionalities: they not only need to be dependable, but also able to adapt to changing situations in their environment. When developing such systems, however, it is often impossible to explicitly design for all potential situations upfront and provide corresponding strategies. Situations that lie out of this "envelope of adaptability" can lead to problems that end up by applying an emergency fail-safe strategy to avoid complete system failure. The existing approaches to self-adaptation cannot typically cope with such situations better—while they are adaptive (and can apply learning) in choosing a strategy, they still rely on a pre-defined set of strategies not flexible enough to deal with those situations adequately. To alleviate this problem, we propose the concept of meta-adaptation strategies, which extends the limits of adaptability of a system by constructing new strategies at runtime to reflect the changes in the environment. Though the approach is generally applicable to most approaches to self-adaptation, we demonstrate our approach on IRM-SA—a design method and associated runtime model for self-adaptive distributed systems based on component ensembles. We exemplify the meta-adaptation strategies concept by providing three concrete meta-adaptation strategies and show its feasibility on an emergency coordination case study.

• Computer systems organization→Embedded and cyber-physical systems • Software and its engineering→ Software organization and properties→ Software system structures→ Software architectures.

Additional Key Words and Phrases: Meta-adaptation strategies, adaptation strategies

1. INTRODUCTION

The advent of cost-effective embedded devices with increased computing capabilities and the proliferation of wireless networks have brought about the potential for value-added services provided by a web of distributed interacting elements organized into Cyber-Physical Systems (CPSs) (NIST 2012; ERCIM 2014). Examples include intelligent navigation systems, smart electric grids, and emergency coordination systems. Modern CPSs need to be able to operate in dynamic or even hostile environments and yet remain dependable and efficient.

An important feature of efficient and dependable CPSs is self-adaptivity, i.e., the ability to modify their behavior and/or structure in response to changes in their

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 by Charles University Grant Agency project No. 391115.

Author's addresses: Technische Universität München, Faculty of Informatics, Chair of Software Engineering, Boltzmannstr. 3, Garching, Germany; Charles University in Prague, Faculty of Mathematics and Physics, Department of Distributed and Dependable Systems, Malostranske namesti 25, Prague, Czech Republic.

DOI:http://dx.doi.org/10.1145/0000000.0000000

environment (de Lemos et al. 2013; Salehie and Tahvildari 2009). Self-adaptation in software systems is usually achieved in three fundamental ways: (i) by relying on a detailed application model, e.g., Markov Decision Processes (MDP) (Ghezzi et al. 2013), and employing simulations or other means of state-space traversal to infer the best response of the system, (ii) by identifying control parameters and employing feedback-based control techniques from control theory (Filieri et al. 2015), and (iii) by reconfiguring architecture models, typically with the help of Event-Condition-Action rules—*architecture-based self-adaptation* (Kephart and Chess 2003; Kramer and Magee 2007; Garlan et al. 2004). These three ways have also been used both combined and together with learning and search-based approaches. For example, control theory has been employed in the runtime modification of the probabilities of a MDP (Filieri et al. 2011). Learning-based approaches have been proposed to deduce the impact of adaptation actions at runtime (Elkhodary, Esfahani, and Malek 2010), and to mine the application model from system execution traces (Yuan, Esfahani, and Malek 2014). Genetic algorithms have been used both in the decision-making process of self-adaptive systems (Ramirez et al. 2009), and in choosing between applicable paths of self-adaptation actions (Ramirez et al. 2010).

In CPSs, as large complex distributed systems, the high-level view of architecture-based self-adaptation is generally favored (David et al. 2009; Garlan et al. 2004; Hirsch et al. 2006). Architecture-based self-adaptation techniques typically follow the MAPE-K loop (or reference model) (Kephart and Chess 2003), which separates self-adaptation into four phases: Monitoring activities, Analyzing runtime metrics, Planning strategies, and Execute plan—all based on a shared Knowledge base. Self-adaptation strategies are expressed as actions involving particular architecture reconfigurations; they are applicable under certain conditions in the presence of certain events or situations (S.-W. Cheng, Garlan, and Schmerl 2012; David et al. 2009; Batista, Joolia, and Coulson 2005). Each action can be associated with the satisfaction of one or more system goals (Salehie and Tahvildari 2012), typically quantified via fitness or utility functions (S.-W. Cheng, Garlan, and Schmerl 2012).

At the same time, due to high *external uncertainty* in CPSs, caused, e.g., by hardware failures and temporary network unavailability, anticipating and *explicitly designing for* all potential situations upfront is not viable. As a result, adapting by switching between the available (pre-defined) self-adaptation strategies may be a challenge, since a CPS might arrive in a situation where none of these strategies applies well, so that a generic "fail-safe" strategy keeping the system operational at a degraded state is the only option.

As a remedy, we propose to generate new self-adaptation strategies at runtime to reflect the changes in the environment and increase the overall system utilities, in particular robustness, resilience, safety, performance, and availability. We do so by introducing the concept of *meta-adaptation strategies*, which enriches the adaptation logic of a CPS (thus the "meta" prefix) by systematically generating new self-adaptation strategies that can be employed equally to pre-defined ones within the adaptation loop. This provides a dynamic space of actions and effectively extends the limits of adaptability of the CPS. We present the idea of meta-adaptation strategies and its integration with the "traditional" MAPE-K loop, which follows a three-layer architecture style. On top of this basis, we show three concrete examples of meta-adaptation strategies and demonstrate their applicability.

Although our approach is to a large extent agnostic to a particular adaptation method, we illustrate and evaluate the application of meta-adaptation strategies within an existing architecture-based self-adaptation technology for distributed

dynamic CPSs—DEECo and IRM-SA—in the context of an emergency coordination case study employed as the running example.

The rest of the paper[1] is structured as follows. Section 2 describes the emergency coordination case study and its DEECo and IRM-SA model. Sections 3 describes the meta-adaptation strategies concept and gives examples of three such strategies. Section 4 reports on our experiments with the three meta-adaptation strategies. Section 5 surveys the related work, Section 6 reflects on the implications and the possible extensions of our approach, and Section 7 concludes.

2. RUNNING EXAMPLE AND BACKGROUND

To demonstrate the concept of meta-adaptation strategies, we briefly overview below the running example used in the paper and the DEECo component model and IRM-SA self-adaptation method, which serve as the technological basis we use to exemplify our approach.

2.1 Running Example: Firefighter Coordination Case Study

The firefighter coordination case study is a real-life real-scale case study that has been proposed for the evaluation of distributed self-adaptive systems (Gerostathopoulos et al. 2016). Firefighters belonging to tactical groups are deployed on the emergency field. They communicate via low-power nodes integrated into their personal protective equipment. Each of these nodes is configured at runtime depending on the task assigned to its bearer. For example, a hazardous situation might need closer monitoring of a certain parameter (e.g., temperature).

In the setting of the complete case study, firefighters have to communicate with the officers (their group leaders), who are equipped with tablets; the software running on these tablets provides a model of the current situation (e.g., on a map) based on data measured at and aggregated from the low-power nodes. Parameters measured at each low-power node are *position*, external *temperature*, *battery level,* and *oxygen level*. The data aggregation on the side of the group leaders is done with the intention that each leader can infer whether any of his/her group members is in danger and take strategic decisions.

Such a coordination system has rigorous safety and performance requirements. It needs to operate on top of opportunistic ad-hoc networks, where no guarantees for end-to-end response time exist, with minimum energy consumption, and without jeopardizing its end-users. It also needs to respond to a number of challenging situations such as: What if the temperature sensor starts malfunctioning or completely fails at runtime? What if firefighters are deployed inside a building where GPS readings are not available? What if the communication between members and their leader is lost?

In all these situations, each node has to adapt its behavior according to the latest information available (choose an appropriate strategy). For example, if a firefighter node detects that it is in the situation “indoors”, it has to switch from the strategy of determining the position via the GPS to using an indoors tracking system. Other strategies include increasing the sensing rate in face of a danger or even relying on the nearby nodes for planning when communication with the group leader is lost.

Obtaining an *exhaustive* list of situations that trigger adaptations in the firefighter coordination system is not a realistic option, as the environment is highly dynamic and unpredictable. In order to deal with this large envelope of adaptability, we rather need

[1] This paper is an extension of (Gerostathopoulos et al. 2015).

```
1.  role GroupMember:
2.    missionID, position
3.  role GroupLeader:
4.    missionID, positionMap
5.
6.  component Firefighter features GroupMember
7.    knowledge:
8.      ID = 59
9.        position = {49.040606, 15.093519}
10.     temperature = 45.2
11.     …
12.   process determinePositionFromGPS
13.       out position
14.     function:
15.       position ←GPSSensor.read()
16.     scheduling: periodic( 500ms )
17.     mode: "outdoors"
18.   … /* other process definitions */
19.
20. ensemble PositionExchange:
21.   coordinator: GroupLeader
22.   member: GroupMember
23.   membership:
24.     member.missionID == coordinator.missionID
25.   knowledge exchange:
26.     coordinator.positionMap ← ( member.id, member.position )
27.   scheduling: periodic( 1000ms )
```

Fig. 1. Excerpt from DSL of DEECo component and ensembles of the firefighter coordination system.

to build a system that would dynamically change its behavior by (i) generating new strategies on demand, and (ii) using them in a MAPE-K-style self-adaptation loop.

2.2 Dependable Emergent Ensembles of Components (DEECo)

DEECo is a component model and corresponding framework that is tailored to the needs of self-adaptive CPSs (Bures et al. 2013; Al Ali et al. 2014). It features two basic abstractions: Autonomous *components* forming dynamic goal-driven collaboration groups called *ensembles*. A component contains knowledge (its data) and processes, whose periodic execution results in updates to the knowledge. Components are not bound to each other; they can only indirectly communicate while being a member/the coordinator of an ensemble. The communication takes the form of mapping the coordinator's knowledge fields into the member component's knowledge field (and vice versa)—*knowledge exchange* (Fig. 1, lines 25-26). Not being static, the membership of a component in an ensemble is periodically evaluated at runtime based on the *membership condition* in the ensemble, specified over partial views of the components' knowledge (Fig. 2, lines 23-24). These views are determined by *roles* specified both in components and ensembles (Fig. 2, lines 1-4, 6, 21-22). The periodic evaluation of membership conditions in ensembles, done by monitoring in DEECo runtime, is an embedded means of self-adaptation in DEECo since its effect is the dynamic adaptation of system architecture to the current component states (the semantics of this adaptation complies with the MAPE-K loop idea). Such inherently dynamic architectures built with DEECo are best suited for distributed systems featuring high

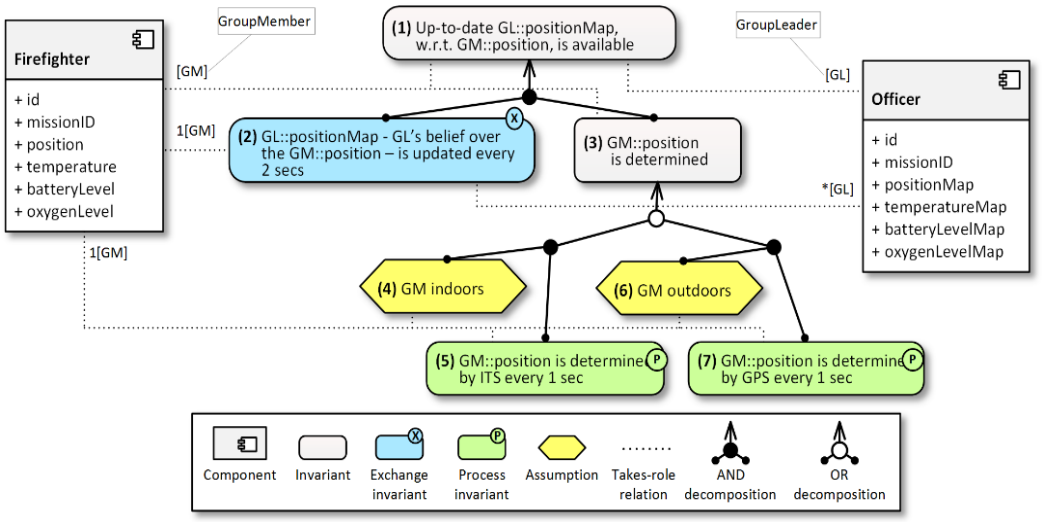

Fig. 2. Excerpt from the IRM-SA model of the running example.

mobility, unstable communication links, and dynamic availability of components in an open environment (such as our running example).

2.3 Invariant Refinement Method for Self-Adaptivity (IRM-SA)

IRM-SA is a requirements-oriented design method tailored for distributed systems of autonomous components communicating in ensembles—thus including DEECo-based systems (Gerostathopoulos et al. 2016; Keznikl et al. 2013). IRM-SA is based on contractual design via iterative refinement of system-level requirements. It captures goals and requirements of the system components as *invariants* that describe the desired state of the system-to-be at every time instant ("invariant" reflects the notion of "continuously striving to achieve a goal", a central notion in highly dynamic systems). An application of the method yields an IRM-SA model represented as a tree. For example, consider invariant (1) in Fig. 2, which specifies that the leader of each firefighter group should have an up-to-date view (encapsulated in the positionMap field) of his/her group members. This "necessity" is AND-decomposed into invariants (2) and (3), which specify the necessities of propagating the position from each member to the leader and determining the position on the site of each member, respectively. The refinement is finished when each leaf invariant of the refinement tree is either an *assumption* or a computation activity corresponding to a *process* or *knowledge exchange* (called *process* and *exchange* invariant, respectively). Alternative designs are captured by the OR-decomposition pattern, where each variant is guarded by an assumption capturing the state of the environment. For example, invariant (3) can be satisfied either by determining the position through an indoors tracking system—invariant (5)—or a global positioning system—invariant (7).

Thus, an IRM-SA model defines a system architecture, by determining which groups (ensembles) of components need to communicate via knowledge exchange and what the internal activity of each component needs to be.

Moreover, employing an IRM-SA model as a model@runtime, the satisfaction of assumptions (4) and (6) can be dynamically monitored to trigger the activity corresponding to the chosen branch in the tree. As a result, the IRM-SA model captures the adaptation logic (strategy) to be used in the planning phase of the MAPE-K loop.

IRM-SA supports the design of DEECo-based systems, since there is a straightforward mapping between the IRM-SA and DEECo constructs: IRM-SA components correspond to DEECo components; process invariants to component processes; exchange invariants to ensembles; and assumptions to DEECo runtime monitors. For example, invariants (7) and (2) of the IRM-SA model of Fig. 2 correspond to the DEECo process and ensemble specification of Fig. 1.

Since the IRM-SA model is also used at runtime in the planning phase of self-adaptation in DEECo-based systems to switch on and off DEECo processes or knowledge exchange functions, an adaptation strategy consists of selecting a set of invariants (a *configuration*) that can satisfy all the root invariants in the model, given the current situation as captured by the assumptions. If no satisfiable configuration can be found, the adaptation fails and thus the system cannot run as intended. Effectively this means that a situation not addressed by the designer has occurred. In the running example, such a case might arise when a firefighter is outdoors and his or her GPS device is malfunctioning. Thus, even though invariant (6) holds and (4) does not, the DEECo process corresponding to invariant (7) cannot be selected because invariant (7) is not satisfiable.

3. META-ADAPTATION STRATEGIES

As already mentioned in the previous sections, due to the large envelope of adaptability, a self-adaptive CPS should ideally devise new strategies to remain operational and satisfy its business goals to the best possible extent. Apart from being able to *devise* new strategies (e.g. by modifying existing ones), the adaptation mechanism should *rank* them according to their effect on the system, in order to be able to select the most promising one, or, at least, to select the ones that are worth trying.

Focusing on architecture-based self-adaptation, we assume that the Business System is adapted via an *Adaptation* Layer (two-layer architecture), instance of the generic MAPE-K loop, implementing a number of adaptation strategies. In our running example, this is done, e.g., by dynamic switching of component processes based on evaluation of the IRM-SA model at runtime. Hence, adaptation strategies are depicted in Fig. 3 as IRM-SA models (subtrees). The selection and execution of an adaptation strategy is the responsibility of the *Adaptation Manager* (similar to Rainbow and Stitch (Garlan et al. 2004; S.-W. Cheng, Garlan, and Schmerl 2012)).

We propose an new architecture style which hoists the handling of situations the system is not explicitly designed for to the architectural level (Fig. 3 and Fig. 3); this three-layer architecture introduces a new architectural entity—*Meta-Adaptation Layer*[2]—which modifies the Adaptation Layer at runtime. Specifically, in our running example, the IRM-SA model is modified at runtime. The "Meta" layer implements a number of meta-adaptation strategies (MTASs), conceptually following the MAPE-K loop by monitoring the effects of the adaptation strategies, analyzing problems (not explicitly-designed-for situations), planning changes (additions/removals/ modifications) to the adaptation strategies, and executing them. As illustrated in Fig. 3, the application of each MTAS results into dynamically creating new adaptation strategies, i.e., IRM-SA models. The selection and execution of a MTAS is the responsibility of the *Meta-Adaptation Manager*.

[2] Similar three-layered architecture has been employed in our recent work on architecture homeostasis (Gerostathopoulos et al. 2016), which, based on the idea of meta-adaptation strategies, illustrates how a concrete system can handle abnormalities and deviations from expected behavior at runtime.

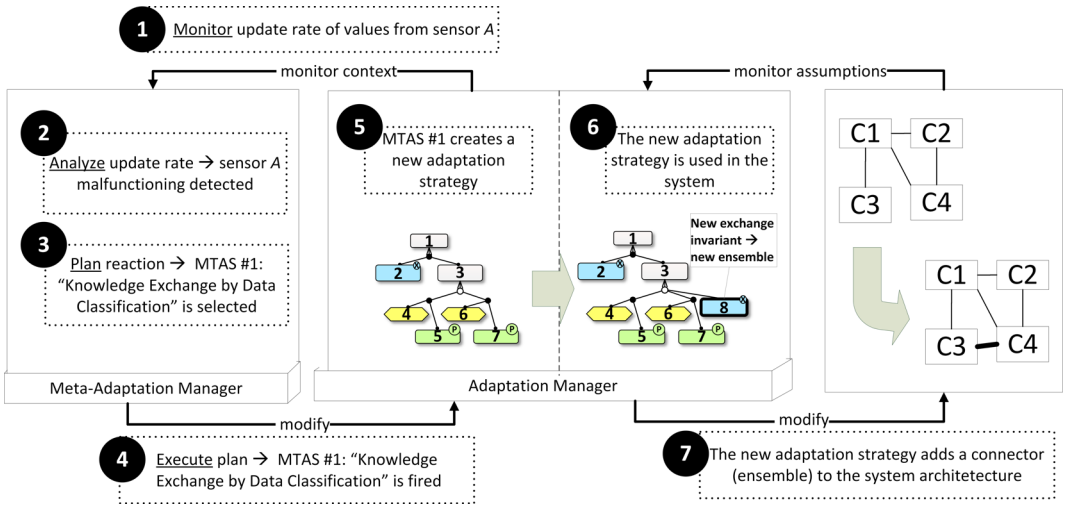

Fig. 4. Steps exemplifying the application of our approach.

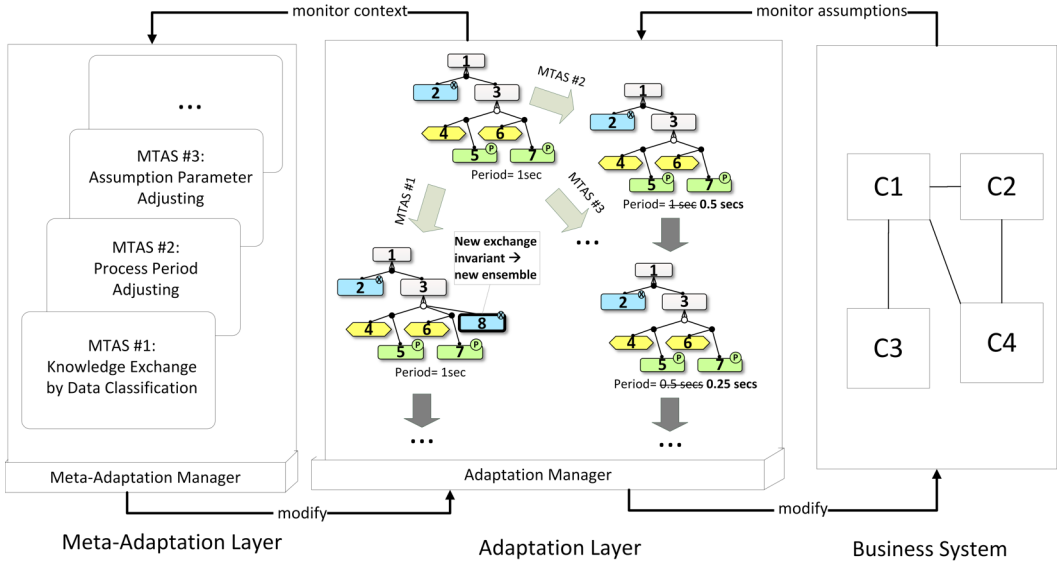

Fig. 3. The three-layer architecture style of our approach. Apart from the original adaptation strategy (top left IRM-SA model at the Adaptation Manager box), three new adaptation strategies are depicted.

In the rest of the section, we detail on the concept of MTASs. MTASs serve as patterns for extending the limits of adaptability of the system, with each MTAS extending the limits in a certain way. The goal of a MTAS is twofold:

1. *To provide an algorithm to systematically generate a set of* strategies *at runtime.*
2. *To provide a way to rank the strategies in the set.*

To facilitate the presentation, understanding, and application of MTASs, we use a template similar in style to the one used by Ramirez et al. to document adaptation design patterns (Ramirez and Cheng 2010) (which was in turn based on the template

Name (abbreviation): A unique handle that describes the MTAS in a succinct way (a code for quick reference).

Intent: The rationale of the MTAS and the goal(s) we want to achieve by applying it.

Context: The non-application-specific conditions in which the MTAS can be applied.

Behavior: Description of the algorithm and/or related UML activity diagram (s) for generating new tactics and the fitness function used for the comparison of the generated tactics.

Trade-offs: List of the trade-offs and possible drawbacks of applying the MTAS.

Example: One or more scenarios that illustrate the problem(s) to be addressed and how the MTAS helps to solve the problem(s).

Fig. 5. Meta-adaptation strategy template.

of Gamma et al. (Erich Gamma et al. 1994)). Fig. 5 depicts the MTAS template and explains each field's intent.

In the rest of the section, we bring our MTAS template into action by presenting three MTASs. Note that the three MTASs can be applied sequentially or in parallel in the running system, since they are by design orthogonal to each other.

3.1 MTAS employing Knowledge Exchange by Data Classification

In CPSs, exploiting the interdependencies between sensed data is an opportunity for introducing specific MTASs. A particular case is the location-dependency of data, i.e., the fact that the value of certain measurable system attributes depends on the physical location of the sensors that provide the data. Below we describe a MTAS providing a way to automatically create knowledge exchange specifications (ensemble specifications in DEECo) that introduce "collaborative sensing" (when direct sensing is not possible anymore) and feed them into the running system. Hence, such new ensembles take the role of new strategies. Fig. 4 provides an overview of the steps in the application of the MTAS.

Name (abbreviation): Knowledge Exchange by Data Classification (DC)

Intent: To increase the robustness of the system by prolonging its acceptable functioning (or by achieving its graceful degradation) in face of data unavailability and outdatedness.

Context: The MTAS targets the case when values of a knowledge field of a component become outdated to the extent that they cannot be relied upon in terms of correct behavior of the component. For instance, there is a sensor malfunction that prevents value updates.

Behavior: To make up for losing the ability to obtain the actual value of an outdated knowledge field, create a new ensemble specification through which the field is assigned an approximated value based on the up-to-date related knowledge values of other components. This specification consists of (a) a membership condition, which

prescribes the condition under the components should interact, and (b) a knowledge exchange function, which specifies the knowledge exchange that takes place between the collaborating components. (For simplicity, we consider the knowledge exchange that just copies the data without manipulating them in any other way.)

To construct the membership condition when the situation targeted by the MTAS happens, the following steps are to be taken:

(i) Observe the system when it is healthy and log components' knowledge (as a time series of the knowledge evolution):

(ii) Analyze (typically offline) the logged knowledge and find conditional correlations indicating that when values of some knowledge fields $C_i.k_l^t, C_j.k_l^t$ are pairwise "close" then other values of $C_i.k_m^t, C_j.k_m^t$ are "close" as well. Here, C_i and C_j are different components, k_l and k_m are different knowledge fields and t is a point in time. If the relation above holds in some specified percentage of all time instances $t = 1..n$, formulate an ensemble (to be instantiated when the situation targeted by the MTAS happened), which uses the pairwise "closeness" of $C_i.k_l, C_j.k_l$ as the membership condition and has the assignment $C_i.k_m \coloneqq C_j.k_m$ as the knowledge exchange.

(iii) Realize (ii) technically by looking for relations $\mu_{k_l}(C_i.k_l^t, C_j.k_l^t) < \Delta_{k_l} \rightarrow \mu_{k_m}(C_i.k_m^t, C_j.k_m^t) < T_{k_m}$ for the time instance $t = 1..n$, which can be established (based on the knowledge observed from the system) on a given confidence level α_{k_m} (e.g, 90% of all the cases). α_{k_m} is specific to each knowledge field k_m. For this, assume user-defined domain-specific distance metrics μ_{k_m} and μ_{k_l}, which provide "distance" between values of knowledge fields k_m and k_l (typically a Euclidean distance), and user-defined domain-specific tolerable distance T_{k_m}, also specific to each knowledge field k_m. Further Δ_{k_l} is a parameter (output of the method), which denotes the maximal distance of knowledge field values that implies correspondence of values $C_i.k_m$ *and* $C_j.k_m$.

(iv) Generate an ensemble with membership condition $\mu_{k_l}(C_i.k_l, C_j.k_l) < \Delta_{k_l}$ and exchange function $C_i.k_m \coloneqq C_j k_m$. This ensemble corresponds to a new strategy. Create more such strategies as more logged data become available.

(v) Select a strategy between the new strategies (ensembles) that update $C_i.k_m$ by applying a fitness function of selecting the strategy which provides the most general membership condition (the one with the largest distance Δ_{k_l}) given the target confidence level α_{k_m}.

Trade-offs: The analysis of the collected time series can be very resource-demanding and therefore a dedicated hardware infrastructure should be used. Similarly, the data collection may be a rather resource-intensive process, especially when components' knowledge is large or changes frequently. Also, introducing superfluous new ensembles can overload the system with unnecessary replicated data.

Example: In the firefighter coordination case study, each firefighter component features the knowledge fields of position and temperature. Suppose that the temperature values are used to control the suit cooling system. Obviously, when the temperature sensor breaks, a real-life threat arises. Since firefighters are usually moving in groups so that those close to each other obtain similar temperature readings, the temperature value of one component can be approximated based on the temperature values of the others, when their positions are close. Technically, the threshold of temperature proximity can be preset (e.g., 20°C).

3.2 MTAS employing Process Period Adjusting

A CPS typically brings real-time requirements that are reflected in schedulability parameters of component processes. The schedulability parameters can be typically inferred by real-time design via schedulability analysis. However, when schedulability parameters influence the systems in a complex manner (e.g., when there is a tradeoff between CPU utilization, battery, network utilization), it is not possible to infer them by systematic analysis. Rather, the schedulability parameters are set manually, based on the experience of the system's architect. The MTAS below addresses the case when the manually set schedulability parameters cannot cope with a situation the CPS is not explicitly designed for.

Name (abbreviation): Process Period Adjusting (PPA)

Intent: To optimize the scheduling of processes with respect to overall system (application-specific) performance in a system where processes are scheduled periodically.

Context: The MTAS targets situations when the system starts failing due to violated timing requirements and the schedulability parameters cannot be inferred a priori because they influence the system in a complex manner.

Behavior: Let R be the set of all active real-time processes in the system. To identify the situation when a requirement for a process r in R with period p is not satisfied anymore, equip each r with a runtime monitor returning a fitness value fv (real number in [0-1]). Generate strategies that correspond to new real-time processes r' created from r by adjusting (reducing or enlarging within pre-defined permissible bounds) p to p', when f drops below a predefined acceptable threshold. To explore the search space of possible period adjustments, employ the genetic algorithm (1+1)-ONLINE EA (Bredeche, Haasdijk, and Eiben 2010) as shown in Fig. 6 (PPA algorithm). Changing p (line 10) can be interpreted as generating a new strategy r' and using it to substitute the strategy r in the system. Terminate the period adjusting procedure when the adjustment of each p has been exercised in both directions and no further benefit can be achieved.

In this MTAS, strategies (new processes) are weighted by applying them to the running system and calculating the overall system fitness as a function of fitness values. The overall fitness is calculated after an observation time window (domain dependent; line 12) allowing several invocations of all processes. For instance, the fitness function in our running example yields the weighted average of functions that measure battery usage, position accuracy and temperature accuracy (Appendix I).

Trade-offs: Reducing periods (a usual action) may have a negative impact on other resources (CPU, battery, network). In such a case, the impact would have to be modelled and taken into consideration in the state-space search.

Example: Consider extending the design of our running example by a root invariant that specifies that "battery consumption should be kept minimized". In order to satisfy this invariant, the system will try at runtime to tweak the processes' periods to invoke them as scarcely as possible. At the same time, when there is high inaccuracy in the GPS readings (e.g., less than 3 satellites in sight), the GPS process may need to be invoked more often to make sure the cumulative inaccuracy of the estimated position of a moving firefighter is within certain bounds. (The cumulative inaccuracy is essentially the sum of the initial inaccuracy of the GPS reading and the distance a

```
1.  begin
2.   foreach Invariant from Processes.Invariants do
3.    Compute fitness for Invariant
4.    OldFit = *CombineFitness(Processes.Invariants.Fitnesses)
5.    Adaptees = *SelectProcessesToAdaptTheirPeriods(Processes)
6.   foreach Process from Adaptees do
7.   begin
8.    *Select the direction for period adjustment (up or down) for Process
9.    *Calculate period delta (difference between old and new period) for Process
10.   Change the period of Process
11.  end
12.  ObserveTime = CalculateObserveTime(Processes)
13.  Run for ObserveTime with no further adaptations for changes to take effect
14.  foreach Invariant from Processes.Invariants do
15.   Compute fitness for Invariant
16.  NewFit = *CombineFitness(Processes.Invariants.Fitnesses)
17.  if NewFit > OldFit then
18.   Keep changes
19.  else
20.   Roll-back changes
21.  end
22. end
```

Fig. 6. Pseudocode capturing a single run of the PPA algorithm. "*" marks steps considered as variation points in the algorithm.

firefighter has moved since the last GPS reading.) It is thus a dynamic trade-off between availability and dependability that has to be resolved at runtime.

3.3 MTAS employing Assumption Parameters Adjusting

In order for a self-adaptive CPS to employ adaptation actions that enable/disable strategies (execute phase of the MAPE-K loop), it first needs to identify the current situation it resides in (monitor and analyze phases). For this, special-purpose activities continuously evaluate the assumptions that each situation corresponds to, regarding the CPS's internal state and its environment. However, these assumptions typically rely on domain knowledge captured by pre-defined behavioral models (e.g., timed automata or state-space models), which may be invalidated at runtime when a CPS reaches a situation it is not explicitly designed for. The MTAS below addresses the case when the manually set parameters of an assumption fail to effectively identify the situation at hand by strengthening or relaxing (similar to the work in (Whittle et al. 2010)) these assumptions. Veritas, a method that uses evolutionary computation to adapt test cases (Fredericks, DeVries, and Cheng 2014) was an inspiration for this MTAS.

Name (abbreviation): Assumption Parameters Adjusting (APA)

Intent: To enhance the monitoring and analysis phases (and therefore prevent premature or delayed actions) by ensuring at runtime that assumptions parameters are relevant to the current situation.

Context: The MTAS targets situations when domain knowledge hardcoded in assumption-monitoring activities fails to frame the current situation, leading, e.g., to very quick or late adaptation actions. This may happen due to a situation which is not modeled by any of the assumptions monitored by the system.

Behavior: Let A be the set of all assumptions in the system. To be able to identify the situation when an assumption a in A is not satisfied anymore, equip each a with a runtime monitor accepting parameters P with values V returning a fitness value fv (real number in [0-1]). Generate (monitoring) strategies that correspond to a new assumption a' created from a by adjusting (reducing or enlarging each parameter value within pre-defined permissible bounds) V to V', when f drops below an acceptable threshold. To explore the search space of possible parameter adjustments, employ the genetic algorithm (1+1)-ONLINE EA (Bredeche, Haasdijk, and Eiben 2010) (similar to the application of the PPA algorithm).

In this MTAS, strategies (new assumptions) are weighted by applying them to the running system and calculating the overall system fitness as a function of fitness values (similar to the PPA fitness function). In our running example, the weighted average of functions that measure battery usage, position accuracy and temperature accuracy was used (Appendix I).

Trade-offs: Relaxing assumptions is not panacea; on the contrary, it can be dangerous in safety critical applications whose operation must not deviate from strict specifications. For this, assumptions that can be relaxed (parametrized) have to be carefully identified at design time, and parameter bounds have to be chosen so that they do not jeopardize the system under any circumstances.

Example: Consider extending the design of our running example by an assumption—sibling to invariant (7): "GM::position is determined by GPS every 1 sec" in Fig. 2—that specifies that "position inaccuracy is within [0,t]", where t is a parameter with values in [5,20] and initial value set to 5. When the inaccuracy of the GPS readings increases (e.g., because of less satellites visible) the assumption will be violated and the activity corresponding to invariant (7) will be deactivated. If the firefighter is outdoors, this will lead to a situation where no configuration is applicable. In response, the system will try to tweak t in order to obtain a new satisfied assumption and consequently an applicable configuration. Though this will yield lower system quality than having the right number of satellites, it will be able to prevent the situation in which the position could not be determined at all due to too strict system design.

4. EXPERIMENTAL EVALUATION

In this section, we describe the experiments performed in order to validate the MTASs idea. The experimental evaluation focuses on measuring the effect of applying the three MTASs to a running system (a scenario from the running example). We first detail the implementation of the three MTASs (Section 4.1), then describe the experimental setting and results (Section 4.2).

4.1 Implementation of MTASs in JDEECo

We have implemented the three MTASs in the JDEECo framework [3], a Java implementation of the DEECo component model. Specifically, each MTAS was implemented[4] as an extension of IRM-SA, an existing JDEECo plugin. IRM-SA endows each component in a DEECo-based application with an adaptation loop. In the nutshell, the IRM-SA adaptation loop periodically (i) aggregates monitoring results from the knowledge of the component and from copies of knowledge of other components (*replicas*), (ii) constructs a runtime representation of the IRM-SA model (cf. Fig. 2), (iii)

[3] https://github.com/d3scomp/JDEECo
[4] https://github.com/d3scomp/IRM-SA/tree/meta-adaptation-strategies

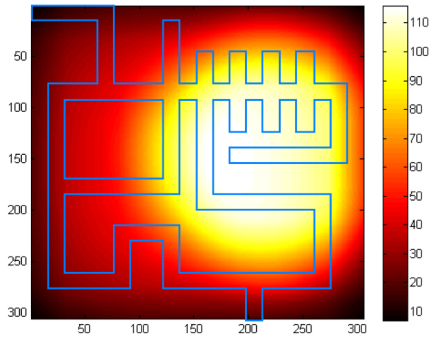

Fig. 7. Map of the example: lines delineate permissible paths for firefighters, background colors depict the temperature (bright colors correspond to high temperatures).

translates the IRM-SA model into a SAT formula, with monitoring variables corresponding to assumptions in the IRM-SA model bound to a particular Boolean value, (iv) invokes the underlying SAT solver and translates its output to the desired current configuration, (v) activates/deactivates component processes and ensemble knowledge exchange processes according to this configuration. For more details on the IRM-SA plugin, we refer the interested reader to (Gerostathopoulos et al. 2016).

When the SAT solving process fails to return an applicable configuration, a dedicated `MetaAdaptationManager` component is notified, which fires up a MTAS. The responsibility of this component is to determine the cause of the reasoning failure (e.g. non-up-to-date data for a knowledge field) and activate the MTAS whose context (cf. Section 3) is closer to the identified cause. Each MTAS is also implemented as a dedicated component with a periodically-invoked `meta-adapt` process.

The DC MTAS is implemented by a dedicated `DCManager` component. `DCManager` aggregates knowledge relevant to DC from all components via a dedicated "aggregation" ensemble, and determines which knowledge values k_m are dependent on other k_l, according to the DC algorithm (Section 3.1). During a run of `DCManager`'s `meta-adapt` process, if a correlation relation is found between knowledge fields k_l and k_m (in components C_i and C_j) with distance of $C_i.k_l$ and $C_j.k_l$ less than Δ_{k_l} and distance of $C_i.k_m$ and $C_j.k_m$ less than T_{k_m} at confidence level α_{k_m}, a new ensemble with membership condition $\mu_{k_l}(C_i.k_l, C_j.k_l) < \Delta_{k_l}$ and exchange function $C_i.k_m := C_j.k_m$ is created and deployed. In further runs, more aggregated knowledge becomes available. As a result, `meta-adapt` may create during its subsequent invocations several such ensembles with different maximal distances Δ_{k_l}, subject to the same confidence level. Since these ensembles are ranked by their maximal distances ${\Delta^1}_{k_l}$, ${\Delta^2}_{k_l}$, etc., `meta-adapt` is responsible for ensuring that always only the "best" of them, i.e., the one with the largest maximal distance, is deployed in the system.

The PPA MTAS is implemented by a dedicated `PPAManager` component. `PPAManager`'s `meta-adapt` process periodically runs the PPA algorithm (Fig. 6). At each run, it selects a set of processes (adaptees) belonging to the local DEECo component and changes their period. So far, we have considered the case where there is only one adaptee – the process with the lowest fitness value. For measuring fitness values, `PPAManager` relies on the output generated by runtime monitors—dedicated periodically-invoked DEECo processes. We have experimented with different values of

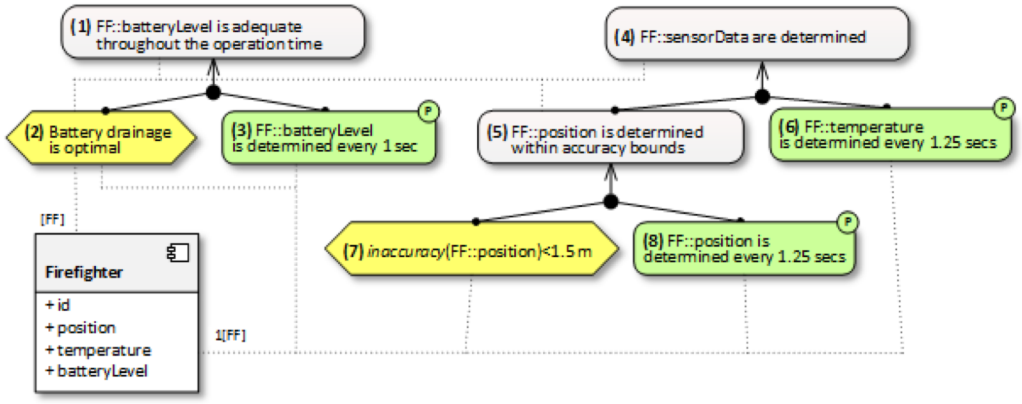

Fig. 8. IRM-SA model of the evaluation example.

parameters of the PPA algorithm (Section 4.2) such as initial direction and the delta of period change. At each run, `meta-adapt` performs one iteration of the PPA algorithm (Fig. 6) and creates a new process by taking the adaptee as blueprint, changing its period, and applying the new process instead of the adaptee to the running system. Then, `meta-adapt` evaluates the new process by observing the system for a time period and calculating the overall fitness of the system over this period. This calculation is a variation point of the PPA algorithm. Other variation points include initial direction and the delta of period change (we have experimented with different setting of these). Further, we used a weighted average over the values retrieved by the runtime monitors. If the overall fitness is reduced when deploying a new process, `meta-adapt` rolls-back the changes and creates another process. This way, new processes are indirectly weighted according to their impact in the running system.

The APA MTAS is implemented by a dedicated `APAManager` component, whose `meta-adapt` process works similarly to `PPAManager`'s `meta-adapt` process. The difference is that, instead of adjusting process periods, it adjusts the parameters of runtime monitors that correspond to assumptions in the IRM-SA model. As in the case of PPA, new assumptions are indirectly weighted (by nature of (1+1)-ONLINE EA) according to their impact in the running system.

4.2 Experimental Setting and Results

Having the implementations of the three MTASs in place, we implemented a simple scenario from the firefighter coordination system (Section 2.1) to acquire initial evidence regarding the feasibility and effectiveness of the three MTASs[5].

Knowledge field	*Distance metric (μ)*	*Tolerable distance (T)*	*Confidence level (α)*
position	Euclidean	4 m	0.9
temperature	difference	20 °C	0.8
battery	difference	0.5 %	0.9

Fig. 9. Distance metrics, tolerable distances, and confidence levels in `Firefighter` knowledge fields.

[5] All the results together with the scripts used to analyze the data can be downloaded from http://d3s.mff.cuni.cz/~skoda/public/TCPS/results.zip

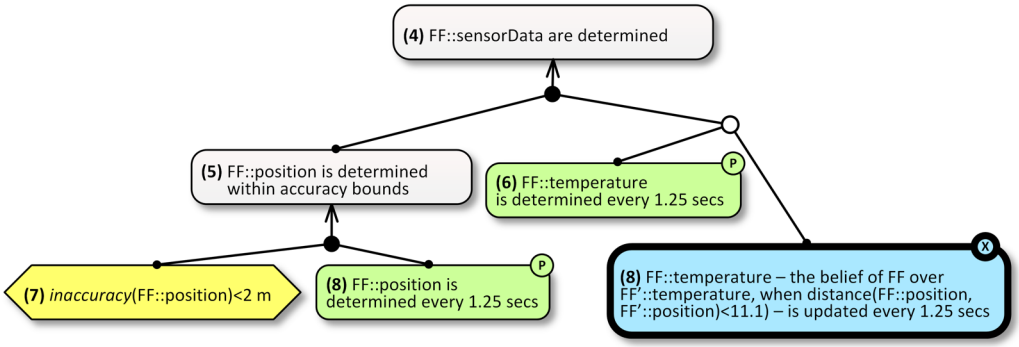

Fig. 10. Final IRM-SA model, after the application of the DC MTAS; the new exchange invariant is depicted in bold.

The example considers three firefighters (`FF1`, `FF2`, and `FF3`) moving in a building (Fig. 7). Each firefighter is represented by a `Firefighter` DEECo component (lines 6-18 in Fig. 1), which senses its position, the surrounding temperature, and the battery level. `FF1` and `FF2` are moving together in a group, whereas `FF3` is moving independently. The objectives of this scenario are represented by the IRM-SA model of Fig. 8.

We used this simple example as testbed for experimenting with the three implemented MTASs. The questions we were mainly aiming to answer were:

1. Does the application of each MTAS help the system deal with situations not explicitly designed for at design time?
2. Can two or more MTASs be used at the same time and result in a positive combined effect to the overall utility of the system?

To answer the above questions, we run simulations of the testbed in JDEECo. In particular, for answering the first question, we run a series of micro-benchmarks (simulation runs lasting 300 seconds each).

To experiment with DC, we used the DC-related metadata depicted in Fig. 9. The values of these metadata were selected based on the domain knowledge of the designer and fine-tuned after few simulation runs. We instructed the simulation environment to introduce a fault in the temperature sensor of `FF1` at the time 150s and observed how the system reacted to this (unexpected at design-time) situation. As expected, shortly after this time instance, `MetaAdaptationManager` was notified that the system cannot self-adapt based on the IRM-SA model. Since `MetaAdaptationManager` could not identify the cause of the fault, it activated all three MTASs simultaneously. Shortly, the `APAManager` and `PPAManager` stopped their execution since their changes had no effect on the system. `DCManager`'s `meta-adapt` process, however, discovered that there is a correlation relation between the distance of the firefighters and their surrounding temperatures (the closer the firefighters are, the more similar the temperature is). It thus deployed a new ensemble that injected the values of the temperature field of `FF3` and `FF2` to the temperature field of `FF1`, whenever `FF2` and `FF3` were close enough to `FF1`. At the time 150s the maximal distance defining the "close enough" was 10.5m. Later in the simulation, a new ensemble with maximal distance 11.1m was created and deployed. Creating these new ensembles corresponds to adjusting the IRM-SA model of the scenario at runtime; the outcome is depicted in Fig. 10. It is important to stress here that the only input we provided for the DC were the distance metrics, tolerable distances, and confidence

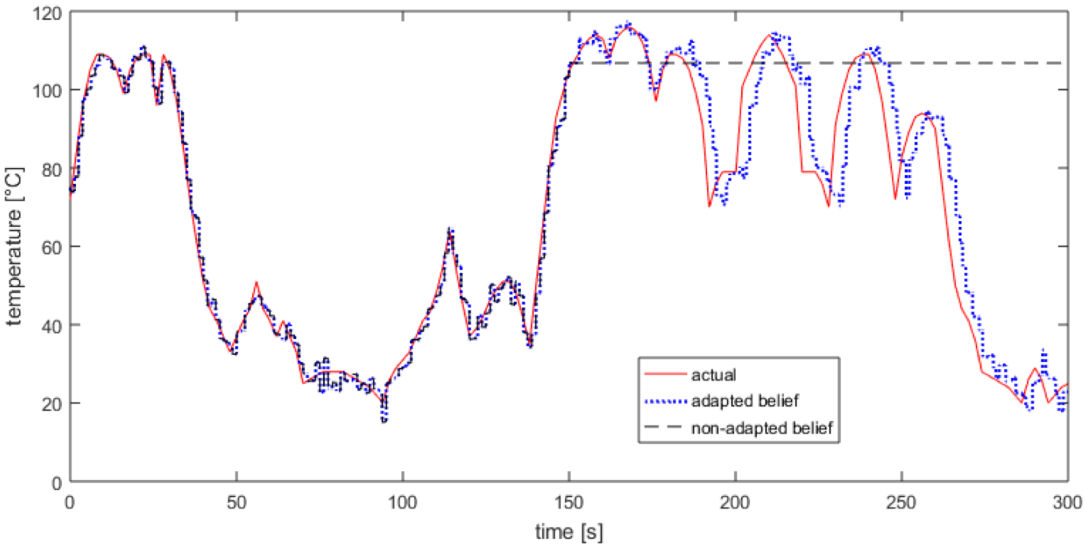

Fig. 11. The evolution of actual and belief temperature of `FF1`.

levels for the three knowledge field of `Firefighter` (Fig. 9). DC automatically detected the correlation and created the new ensemble.

A graphical illustration of the impact of the application of DC is provided in Fig. 11. The solid line represents the actual temperature of `FF1`'s surroundings. The dashed line represents the belief about the temperature in the simulation run that does not employ MTASs (the baseline for the evaluating MTASs). The dotted line represents the belief about the temperature when MTASs are employed. Before time 150s, the belief about the temperature is rather accurate—disturbed only by random noise and movement of the firefighter between temperature sampling. After the temperature sensor stops working (time 150s), when MTASs are not employed, `FF1` obtains no more temperature updates so that the temperature belief becomes obsolete and unreliable (horizontal part of the dashed line). In case MTASs are employed (dotted line), the temperature belief is being updated by the temperature sensed by `FF2` and `FF3`, as explained above. Even though there is a delay and inaccuracy, the temperature belief is usable.

To experiment with PPA and APA (and their combination), we performed another set of simulation runs. We instructed the simulation environment to increase the inaccuracy of position readings of `FF1` at the time 50s. Shortly, `MetaAdaptationManager` was notified that the system cannot self-adapt according to the IRM-SA model as the position inaccuracy it higher than assumed (cf. invariant (7) in Fig. 8). Since the position values were continued to get updated (only with inaccurate readings), `MetaAdaptationManager` did not activate DC, but only PPA and APA. PPA identified invariant (8): "FF:position is determined every 1.25 secs" as a candidate for improvement and decreased the period of `determinePositionFromGPS` in `FF1` (lines 12-17 in Fig. 1). This decrease reduced also the inaccuracy of position values between subsequent readings. This partially mitigated the effect of the increased inaccuracy of the position readings. Notably, there were several runs of the `PPAManager`'s `meta-adapt` process. In each run, a new process with smaller period (reduction by 250ms) than the adaptee was created and deployed. However, the period

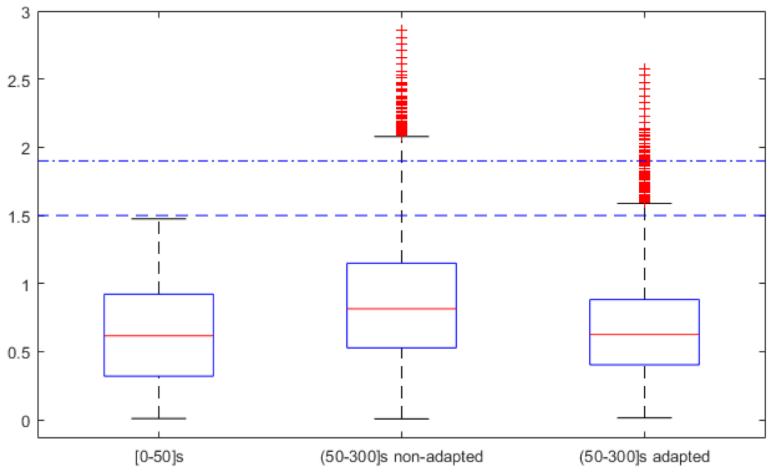

Fig. 12. The distance between actual position and belief position. It is split into the case with and without meta-adaptation, and into the interval before and after the position sensor faults.

could not be reduced below a given lower limit (250ms in this case), which means that PPA itself would not have been able to improve the position inaccuracy back below the required boundary (1.5 m). This situation was addressed by APA, which identified assumption (7) (Fig. 8) as a candidate for relaxation and increased the threshold of the allowed position inaccuracy (while still keeping it within designated bounds). These two changes finally resulted in a system state valid with respect to the IRM-SA specification.

Fig. 12 shows the influence of PPA and APA on the system after the malfunction occurs at the time 50s. The boxplots depict the difference between the actual position of `FF1` and its position belief (sampled every 1250ms). The left-most boxplot depicts the baseline. The middle and right-most boxplots depict the data obtained in the interval 50s to 300s. While the middle boxplot corresponds to not employing MTASs, the right-most indicates the effect of MTASs application. Clearly, the increased inaccuracy of the position readings increased the difference between the actual and believed position; using the PPA and APA helped mitigate this problem. The 2-sample t-test that we performed on the samples depicted in the middle and right-most boxplot showed (at confidence interval 95%) a statistical significant difference with p-value < 2.2e-16. The horizontal dashed and dotted-dashed lines denote the assumed limit on position inaccuracy—the former corresponds to the original limit (1.5 m), the latter to the limit relaxed by the APA MTAS (1.9 m).

5. RELATED WORK

Self-adaptation has been a flourishing research topic within the software engineering community (B. Cheng, de Lemos, et al. 2009; de Lemos et al. 2013; Salehie and Tahvildari 2009) and has been primarily tackled by (i) modeling and model-driven engineering (Zhang and Cheng 2006; Goldsby and Cheng 2008; Morandini and Perini 2008), (ii) control theory (Patikirikorala et al. 2012; Filieri et al. 2011; Filieri, Hoffmann, and Maggio 2014; Filieri et al. 2015), and (iii) software architecture (Garlan et al. 2004; S.-W. Cheng, Garlan, and Schmerl 2012; Sykes et al. 2008; Kramer and Magee 2009).

Although tailored to architecture-based self-adaptation, the proposed MTASs concept can be used to complement different approaches from (i)-(iii) to extend the limits of adaptability of a system. For example, if a model-driven approach is able to generate new behavior models and select among them at runtime, there is a good chance that it can be articulated as a MTAS. AVIDA-MDE stands as a good example towards this direction (Goldsby and Cheng 2008). Therein, a MTAS takes the form of generating behavioral models (UML state diagrams) via a digital evolution-based approach, backed by an evolutionary computation platform (Ofria and Wilke 2004). In AVIDA-MDE, a MTAS fitness function is expressed in terms of both the non-functional characteristics and the latent functional properties that the generated models exhibit. Although AVIDA-MDE is an offline approach, its importance lies in the automatic generation of strategies, which the MTASs idea relies upon. Methods of synthetizing computationally diverse program variants (Baudry et al. 2014) have been similar sources of our inspiration.

To deal with uncertainty at runtime, Veritas, a method that uses evolutionary computation to adapt test cases (Fredericks, DeVries, and Cheng 2014) was an inspiration for our APA MTAS. The main idea of it is to apply (1+1)-ONLINE algorithm to generate new test cases that better reflect the changes in the behavior of a self-adaptive system, i.e., produce less false negatives when exercised in the running system. In comparison, a MTAS takes the form of generating new test cases by iterative application of the evolutionary algorithm. The overall system fitness is measured by fitness functions (similarly to our APA MTAS) and can be interpreted as the fitness function of APA. Veritas has been integrated in a framework that addresses assurance at run time in face of uncertainty called Proteus (Fredericks and Cheng 2015). Apart from online adaptive testing mentioned above, component-based integration testing has also been advocated to support runtime compliance checking of self-adaptive systems (da Silva and de Lemos 2011).

Extending the envelope of adaptability of a system in order to deal with uncertainty (that effectively "drives" self-adaptation) has been pursued at different levels. Important ideas have been spurred at the requirements engineering community in approaches such as FLAGS (Baresi and Pasquale 2010; Baresi, Pasquale, and Spoletini 2010), Evolution Requirements (Souza, Lapouchnian, and Mylopoulos 2012; Souza et al. 2012), and the combination of KAOS with RELAX (B. Cheng, Sawyer, et al. 2009).

FLAGS focuses on modeling "adaptation requirements", i.e., such requirements that concern countermeasures to be taken when application requirements fail. FLAGS relies on the goal, operation, and object models of KAOS (Lamsweerde 2008) and extends them. The main concept is an *adaptation goal*, a special type of goal that corresponds to a countermeasure to be applied when a "conventional" goal fails to meet its satisfaction criteria. To model such criteria, FLAGS introduces the notion of fuzzy goal. For the formal specification of fuzzy goals, RELAX language (Whittle et al. 2010) was employed (Baresi and Pasquale 2011). According to the degree of satisfaction of goals at runtime, adaptation goals can be triggered, resulting into counteractions that include adding/removing goals, operations, or objects, modifying the membership function of goals, and adjusting the pre- and post-conditions of operations (Baresi, Pasquale, and Spoletini 2010).

Evolution Requirements (EvoReqs) focuses on modeling the requirements that cause the evolution of other requirements. It is model-based; since (traditional) requirements are modeled as goals, evolution requirements are then modeled as event-condition-action rules that refer to the events acting as guard conditions for the goals. The main idea is that when a requirement cannot be satisfied anymore, it is changed.

Examples of changes include retrying after some time, relaxing the requirement, delegating it to a human actor, and replacing a domain assumption with a system task (Souza et al. 2012).

The main differences of FLAGS and EvoReqs to our MTASs idea is that (i) they focus primarily on requirements specification, while our approach focuses more on runtime behavior (although, in many cases, important design effort needs to be furnished, cf. Section 6.5), and (ii) they cannot handle situations not explicitly designed for, since each situation and corresponding strategy has to be modeled upfront.

Another noteworthy approach to deal with environmental uncertainty at the requirements level is the combination of KAOS method with the RELAX language (B. Cheng, Sawyer, et al. 2009). In this approach, goals are first elicited and decomposed forming a KAOS model (Lamsweerde 2008). Then obstacles/threats are identified, corresponding to environmental conditions that pose uncertainty at development time and may warrant dynamic adaptation at runtime. Finally, "mitigation tactics" are devised in order to deal with the identified threats. These tactics take the form of adding low-level sub-goals, relaxing a goal (adding flexibility to goal satisfaction using RELAX), or adding a high-level goal. We view the above approach complementary to ours: While they focus on identifying application-specific threats that are anticipated at development time and on providing means to mitigate them, we focus on identifying domain-specific contexts and generating and ranking runtime strategies that are to be triggered in such contexts.

A discussion on the four different levels of requirements engineering for and in self-adaptive systems (Berry, Cheng, and Zhang 2005) acknowledges the difficulty in developing self-adaptive systems able to adapt to changes not explicitly designed for at runtime; our MTASs idea is a step towards this direction. With respect to this four-level classification, we view the MTASs as an enhancement of Level 3 RE, since even though "they are designed by humans" they determine adaptation elements in an automated way.

Finally, a three-layer architecture for evolution of dynamically adaptive systems has been proposed also by (Perrouin et al. 2012). Contrary to their work, which focuses on using an evolution layer for switching between available self-adaptation strategies, we propose the top layer to create new self-adaptation strategies at runtime. In general, our approach 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.

6. DISCUSSION

6.1 *Problem Space Coverage*

Generally, if a system is subject to environment uncertainty, the extent of the problem space that should be covered by system's adaptability is unknown. This makes it impossible to devise all adaptation strategies at design time. It of course makes it also impossible to presume all necessary MTASs, since each MTAS covers only a certain sub-space of the problem space. However, compared to pre-designed strategies, the MTASs involves observation of system's and environment's evolution at runtime and utilizes this to formulate new strategies. As such, it has the potential to carry through higher expressive power than pre-designed strategies and consequently achieve higher coverage of the problem space. Moreover, it is possible to combine MTASs with orthogonal intent, as we illustrated in Sect. 4, and thus to increases the problem space coverage even further.

6.2 *Interplay of* Meta-Adaptation Strategies

We assume that MTASs may overlap in their actions. This brings up the question of their combined effect. Specifically, the interesting case is when more MTASs generate two strategies that the adaptation manager can exploit at the same time. In the running example, this is illustrated by the interplay of PPA and APA to address the problem of low inaccuracy of position readings. PPA creates a strategy of sensing position more frequently, while APA creates a strategy of relaxing the requirements on inaccuracy. It is the responsibility of the meta-adaptation mechanism to strike a balance between the overlapping strategies. For instance, in our running example, this is achieved by associating APA with a priority lower than PPA, which effectively applies APA only when PPA hits its limits (a period cannot be lowered any further).

In general, each MTAS has an impact on the cost and risk in the system—e.g., more frequent sensing of position increases power consumption, while relaxing the requirements may increase the risk that the system does not promptly react to changing conditions. To address related trade-offs we assume fitness functions associated with system goals (invariants in IRM-SA) will be provided by the user and employed by the meta-adaptation mechanism (e.g. by `MetaAdaptationManager`). In a similar vein, the meta-adaptation mechanism might periodically check the system's performance and cost and propose MTASs that can optimize them—e.g., APA may strengthen a previously relaxed requirement. Nevertheless these issues are out of the scope of this paper.

6.3 Generality of the approach

Though the combination of IRM-SA and DEECo was used in this paper to demonstrate the concept of MTASs, there are other approaches to self-adaptation that can be extended by MTASs. An option is to extend Rainbow (Garlan et al. 2004) and Stitch (S.-W. Cheng, Garlan, and Schmerl 2012). Stitch is a language for specifying adaptation strategies, tailored for the domain of system administration. The strategy concept in Stitch encompasses tactics, which in turn consist of operators as system-provided configuration commands (e.g., "start new virtual machine"). A tactic in Stitch is a specification of an activity with a pre- and post-condition and an associated action, while a strategy is a specification of a process, each step of which involves the conditional execution of a tactic. In this respect, a Stich strategy loosely corresponds to an IRM-SA model (but not to a MTAS, since it does not generate new strategies), while a Stitch tactic to an IRM-SA invariant. Given these facts, embedding the MTASs concept into Stitch is a viable and promising option.

All in all, the concept of hoisting the handling of situations the system is not explicitly designed for at a separate architecture layer is fairly general. The particular MTASs are domain-specific and as a result, they can be applied to different applications of the same domain—CPS. In our recent work, we applied them to a robotic scenario featuring different CPS challenges (Gerostathopoulos et al. 2016).

6.4 Other Meta-Adaptation Strategies

In addition to the three MTASs introduced in Section 3 we describe here initial ideas on the formulation of additional MTASs, stemming from our consideration when it is both desirable and realistic to create new strategies at runtime.

For instance, an idea for a potential MTAS is to allow a component to perform a process that is originally designed for a different component with similar knowledge. While from the perspective of a single component this is providing a way to extend its set of available strategies, from a system perspective this can be seen as a possibility

for runtime optimization of the data flow in the system. Along these lines, a "Data Flow Adjusting" (DFA) MTAS would be responsible for obtaining a system level view and instructing individual components to apply novel strategies that have the potential to optimize system-level properties such as latency and performance. For example, DFA could be useful in a case where component A cannot communicate with component B anymore (e.g., because of unreliable connection), which results into violating certain coordination requirements (captured by exchange invariants in IRM-SA). In such a case, the component A might obtain adequate data from a component C, or even create a process that senses the missing data directly.

Another idea for a potential MTAS is rooted in the observation that traditional self-adaptation approaches operating at a single component rely on sensing specific data either directly or indirectly (via communication with other components). When a faulty sensor starts emitting wrong data, a "Faulty Sensor Isolation" (FSI) MTAS could instruct the components that consume the wrong data to disregard it. In the context of IRM-SA and DEECo, FSI would take the form of creating a new ensemble that copies only the subset of "healthy" data from the component A (with the malfunctioning sensor) to component B, and using this new ensemble to replace the existing one mediating so far the communication of A and B.

Self-adaptation is also often specified as switching between operational modes of components in a component-based system. Such switching is governed by mode guards and is usually deterministic. Nevertheless, having a MTAS that can introduce probabilistic mode switching and tryout different transition probabilities, holds the potential of more flexible reactions to corner cases where components get stuck into their over-constrained behavior specified by too rigid mode switching.

Finally, a potential MTAS idea comes from the straightforward extension of PPA to consider adjusting not only the periods of component processes, but also the periods of communication activities (exchange invariants in IRM-SA, such as invariant (2) in Fig. 2). In such a case, an "Exchange Period Adjusting" (EPA) MTAS would modify the communication activities and, at the same time, their effect would have to be evaluated via calculating the overall system fitness in a similar vein as in PPA and APA.

6.5 A Classification Scheme for Meta-Adaptation Strategies

In order to examine our experience with MTASs so far and systematize the efforts for deriving additional MTASs, we detail here a set of dimensions that form a preliminary classification scheme for MTASs (Fig. 13). The concepts of adaptation elements of detection and monitoring techniques, decision-making procedures (Berry, Cheng, and Zhang 2005) were used as inspiration for the first two dimensions. As to adaptive mechanisms (also Berry, Cheng, and Zhang 2005), all MTASs considered in this paper rely on architecture-based adaptation mechanisms, so that we do not explicitly include the adaptive mechanisms dimension in Fig. 13.

Detection and monitoring techniques. The first step in meta-adapting a self-adaptation mechanism is to determine the context of meta-adaptation, that is, the class of situations that warrant meta-adaptation in the system (*MTAS context*). It is important that we are not interested in identifying the situations themselves, but rather their class, since MTASs are (or should be) specific to a domain and not a specific application. In a naïve approach, we can advocate that the system should meta-adapt simply when the existing self-adaptation mechanism fails to deliver its goal, i.e., to adjust the system towards meeting its requirements. A key issue is in which situations does this happen, and, more importantly, what needs to be monitored in order to determine the class of such problematic situations at runtime.

Meta-Adaptation Strategy	*Detection and monitoring techniques*	*Decision-making procedures*		*Applicability attributes*
		Amount of data	*Amount of design effort*	
Knowledge Exchange by Data Classification (DC)	localized	large	moderate	data correlation infrastructure
Process Period Adjusting (PPA)	localized	moderate	moderate	time-scheduled processes
Assumption Parameters Adjusting (APA)	localized	moderate	moderate	domain knowledge captured by behavioral models
Data Flow Adjusting (DFA)	system-level	large	moderate	data flow model and optimization criteria
Faulty Sensor Isolation (FSI)	system-level	little	almost none	reliability models of sensors
Exchange Period Adjusting (EPA)	system-level	moderate	moderate	time-scheduled communication

Fig. 13. Positioning of MTASs introduced in this paper in the tentative classification scheme comprised of three dimensions. All these MTASs rely on architecture-based adaptation mechanisms.

In some MTAS contexts, the source of the fault is localized, so detection and identification can be localized as well. This is for example the case of DC; the situations belonging to the context of DC can be identified by looking at the outdatedness of data sensed by sensors. If the sensed data is outdated, there is evidence that the sensor stopped functioning. In MTAS contexts, data from different parts of a system need to be monitored, as the source of fault is not localized. This is, e.g. the case of DFA context: Monitoring a single component/sensor does not typically suffice in detecting a poorly performing data-flow architecture. Needless to say, system-level monitoring puts forward more design and operational challenges for a MTAS w.r.t. its overhead in the system.

Decision-making procedures. Once a MTAS context is identified, the corresponding MTAS is activated (assuming at least one exists). A decision-making procedure then takes place in order to create new strategies at runtime and choose the most promising ones to add to the underlying self-adaptation mechanism. This procedure differs from among MTASs w.r.t to (i) the amount of data that needs to be collected, and (ii) the amount of effort that needs to be invested upfront in the configuration/customization of the MTAS's decision-making procedure by the system designer.

For example, DC needs to collect an extensive amount of data in order to create new ensembles on-the-fly, and a considerable design effort to provide appropriate domain-specific distance metrics, tolerable distances, and confidence levels for each knowledge field in the system (cf. Fig. 9). Both PPA and APA need to collect a moderate amount of data to determine the adaptees and to measure the overall system fitness. Moreover, design effort is needed in choosing between the variation points of the PPA and APA algorithms (Fig. 6). On the other hand, FSI does not need to collect a lot of data as its decision-making is straightforward once a fault has been identified; no design effort is needed either.

Applicability attributes. Each MTAS makes a number of assumptions regarding the underlying system. These assumptions form the perimeter of applicability of each MTAS. For example, DC assumes that a data correlation infrastructure is available. PPA, resp. EPA, assume that there are some time-scheduled processes, resp. communication, in the system. APA assumes that assumptions modeling and monitoring relies on domain knowledge captured by timed automata or other behavioral models. DFA assumes that there is a data flow model of the whole system with inputs and outputs for each component. Finally, FSI assumes that the reliability of sensing devices is modeled a priori (so that incorrect output can be detected).

7. CONCLUSION

In this paper, we have focused on the problem of dealing with the large number of situations and corresponding self-adaption strategies which form the envelope of adaptability in self-adaptive cyber-physical systems. We claimed that explicitly designing for all the different situations is not viable and proposed to generate new self-adaptation strategies at runtime to reflect the changes in the environment. Our approach takes the form of a three-layer architectural style which hoists the handling of situations the system is not explicitly designed for to the architectural level. The top layer of our approach consists of a number of meta-adaptation strategies which collectively enhance the envelope of adaptability of a system.

In addition to laying out the general concept of meta-adaptation strategies, we have exemplified the concept by three rather diverse such strategies: Knowledge Exchange by Data Classification, Process Period Adjusting, and Assumptions Parameters Adjusting. We have implemented the proposed approach on top of IRM-SA and DEECo as plugins to the IRM-SA and provided an evaluation which shows that a combination of such meta-adaptation strategies has indeed the result of extending the system's adaptability and robustness.

The meta-adaptation strategies presented in this paper (including the preliminary ones discussed in Sections 6.4 and 6.5) do not cover the whole space of potential meta-adaptation strategies. We also note that even though the application of different meta-adaptation strategies allows for systematic gradual design via separation of concerns, it has to be carefully designed and evaluated. Finally, we believe that by introducing the idea of meta-adaptation strategies as means for dynamically extending the limits of system's adaptability we provide helpful inspiration for future research on self-adaptive systems.

REFERENCES

Al Ali, Rima, Tomas Bures, Ilias Gerostathopoulos, Petr Hnetynka, Jaroslav Keznikl, Michal Kit, and Frantisek Plasil. 2014. "DEECo: An Ecosystem for Cyber-Physical Systems." In *Companion Proc. of ICSE '14*, 610–11. ACM Press. doi:10.1145/2591062.2591140.

Baresi, Luciano, and Liliana Pasquale. 2010. "Live Goals for Adaptive Service Compositions." In *Proc. of SEAMS '10*, 114–123. SEAMS '10. New York, NY, USA: ACM. doi:10.1145/1808984.1808997.

———. 2011. "Adaptation Goals for Adaptive Service-Oriented Architectures." In *Relating Software Requirements and Architectures*, edited by Paris Avgeriou, John Grundy, Jon G. Hall, Patricia Lago, and Ivan Mistrík, 161–81. Springer Berlin Heidelberg. http://link.springer.com/chapter/10.1007/978-3-642-21001-3_10.

Baresi, Luciano, Liliana Pasquale, and Paola Spoletini. 2010. "Fuzzy Goals for Requirements-Driven Adaptation." In *Proc. of RE '10*, 125–34. IEEE. doi:10.1109/RE.2010.25.

Batista, Thais, Ackbar Joolia, and Geoff Coulson. 2005. "Managing Dynamic Reconfiguration in Component-Based Systems." In *Software Architecture*, edited by Ron Morrison and Flavio Oquendo, 1–17. Lecture Notes in Computer Science 3527. Springer Berlin Heidelberg. http://link.springer.com/chapter/10.1007/11494713_1.

Baudry, B., M. Monperrus, C. Mony, F. Chauvel, F. Fleurey, and S. Clarke. 2014. "DIVERSIFY: Ecology-Inspired Software Evolution for Diversity Emergence." In *Proc. of CSMR-WCRE '14*, 395–98.

doi:10.1109/CSMR-WCRE.2014.6747203.

Berry, Daniel M, Betty H C Cheng, and Ji Zhang. 2005. "The Four Levels of Requirements Engineering for and in Dynamic Adaptive Systems." In *Proc. of the 11th International Workshop on Requirements Engineering Foundation for Software Quality, Porto, Portugal*, 95–100.

Bredeche, N., E. Haasdijk, and A. E. Eiben. 2010. "On-Line, On-Board Evolution of Robot Controllers." In *Artifical Evolution*, edited by Pierre Collet, Nicolas Monmarché, Pierrick Legrand, Marc Schoenauer, and Evelyne Lutton, 110–21. Lecture Notes in Computer Science 5975. Springer Berlin Heidelberg. http://link.springer.com/chapter/10.1007/978-3-642-14156-0_10.

Bures, Tomas, Ilias Gerostathopoulos, Petr Hnetynka, Jaroslav Keznikl, Michal Kit, and Frantisek Plasil. 2013. "DEECo – an Ensemble-Based Component System." In *Proc. of CBSE'13*, 81–90. ACM. doi:10.1145/2465449.2465462.

Cheng, Betty, Rogerio de Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, et al. 2009. "Software Engineering for Self-Adaptive Systems: A Research Roadmap." In *Software Engineering for Self-Adaptive Systems*, 1–26. Springer Berlin Heidelberg. http://www.springerlink.com/index/H380742725036312.pdf.

Cheng, Betty, Pete Sawyer, Nelly Bencomo, and Jon Whittle. 2009. "A Goal-Based Modeling Approach to Develop Requirements of an Adaptive System with Environmental Uncertainty." In *Proc. of MODELS '09*, 1–15. Springer Berlin Heidelberg. doi:10.1007/978-3-642-04425-0_36.

Cheng, Shang-Wen, David Garlan, and Bradley Schmerl. 2012. "Stitch: A Language for Architecture-Based Self-Adaptation." *Journal of Systems and Software* 85 (12): 1–38. doi:10.1016/j.jss.2012.02.060.

da Silva, Carlos Eduardo, and Rogério de Lemos. 2011. "Dynamic Plans for Integration Testing of Self-Adaptive Software Systems." In *Proc. of SEAMS '11*, 148–157. SEAMS '11. New York, NY, USA: ACM. doi:10.1145/1988008.1988029.

David, Pierre-Charles, Thomas Ledoux, Marc Léger, and Thierry Coupaye. 2009. "FPath and FScript: Language Support for Navigation and Reliable Reconfiguration of Fractal Architectures." *Annals of Telecommunications* 64 (1–2): 45–63. doi:10.1007/s12243-008-0073-y.

de Lemos, Rogerio, Holger Giese, Hausi A. Muller, Mary Shaw, Jesper Andersson, Marin Litoiu, Bradley Schmerl, et al. 2013. "Software Engineering for Self-Adaptive Systems: A Second Research Roadmap." In *Software Engineering for Self-Adaptive Systems II*, 7475:1–32. LNCS. Springer Berlin Heidelberg. http://link.springer.com/chapter/10.1007/978-3-642-35813-5_1.

Elkhodary, Ahmed, Naeem Esfahani, and Sam Malek. 2010. "FUSION: A Framework for Engineering Self-Tuning Self-Adaptive Software Systems." In *Proc. of FSE '10*, 7–16. FSE '10. ACM. doi:10.1145/1882291.1882296.

ERCIM. 2014. "Special Theme: Cyber-Physical Systems," April.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional.

Fairbanks, George. 2014. "Architectural Hoisting." *IEEE Software* 31 (4). doi:http://doi.ieeecomputersociety.org/10.1109/MS.2014.82.

Filieri, Antonio, Carlo Ghezzi, Alberto Leva, Martina Maggio, and Politecnico Milano. 2011. "Self-Adaptive Software Meets Control Theory: A Preliminary Approach Supporting Reliability Requirements." In *Proc. of ASE '11*, 283–92. IEEE. doi:10.1109/ASE.2011.6100064.

Filieri, Antonio, Henry Hoffmann, and Martina Maggio. 2014. "Automated Design of Self-Adaptive Software with Control-Theoretical Formal Guarantees." In *Proc. of ICSE '14*, 299–310. ACM Press. doi:10.1145/2568225.2568272.

Filieri, Antonio, Martina Maggio, Konstantinos Angelopoulos, Nicolas D'Ippolito, Ilias Gerostathopoulos, Andreas Berndt Hempel, Henry Hoffmann, et al. 2015. "Software Engineering Meets Control Theory." In *Proc. of SEAMS '15*, 71–82. IEEE. http://dl.acm.org/citation.cfm?id=2821357.2821370.

Fredericks, Erik M., and Betty H. C. Cheng. 2015. "Automated Generation of Adaptive Test Plans for Self-Adaptive Systems." In *Proc. of SEAMS '15*, 157–68. IEEE. http://dl.acm.org/citation.cfm?id=2821357.2821385.

Fredericks, Erik M., Byron DeVries, and Betty H. C. Cheng. 2014. "Towards Run-Time Adaptation of Test Cases for Self-Adaptive Systems in the Face of Uncertainty." In *Proc. of SEAMS '14*, 17–26. ACM Press. doi:10.1145/2593929.2593937.

Garlan, David, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. 2004. "Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure." *Computer* 37 (10): 46–54. doi:10.1109/MC.2004.175.

Gerostathopoulos, Ilias, Tomas Bures, Petr Hnetynka, Adam Hujecek, Frantisek Plasil, and Dominik Skoda. 2015. "Meta-Adaptation Strategies for Adaptation in Cyber-Physical Systems." In *Proc. of ECSA '15*, 45–52. Springer. doi:10.1007/978-3-319-23727-5_4.

Gerostathopoulos, Ilias, Tomas Bures, Petr Hnetynka, Jaroslav Keznikl, Michal Kit, Frantisek Plasil, and Noël Plouzeau. 2016. "Self-Adaptation in Software-Intensive Cyber–physical Systems: From System Goals to Architecture Configurations." *Journal of Systems and Software*. Accessed June 15. doi:10.1016/j.jss.2016.02.028.

Gerostathopoulos, Ilias, Dominik Skoda, Frantisek Plasil, Tomas Bures, and Alessia Knauss. 2016. "Architectural Homeostasis in Self-Adaptive Software-Intensive Cyber-Physical Systems." In *In Proc. of the 10th European Conference on Software Architecture, to Appear*.

Ghezzi, Carlo, Leandro Sales Pinto, Paola Spoletini, and Giordano Tamburrelli. 2013. "Managing Non-Functional Uncertainty via Model-Driven Adaptivity." In *Proc. of ICSE'13*, 33–42. ICSE '13. IEEE. http://dl.acm.org/citation.cfm?id=2486788.2486794.

Goldsby, HJ, and BHC Cheng. 2008. "Automatically Generating Behavioral Models of Adaptive Systems to Address Uncertainty." In *Proc. of MODELS'08*, 568–83. Springer Berlin Heidelberg. doi:10.1007/978-3-540-87875-9_40.

Hirsch, Dan, Jeff Kramer, Jeff Magee, and Sebastian Uchitel. 2006. "Modes for Software Architectures." In *Software Architecture*, edited by Volker Gruhn and Flavio Oquendo, 113–26. Lecture Notes in Computer Science 4344. Springer Berlin Heidelberg. http://link.springer.com/chapter/10.1007/11966104_9.

Kephart, Jeffrey, and David Chess. 2003. "The Vision of Autonomic Computing." *Computer* 36 (1): 41–50.

Keznikl, Jaroslav, Tomas Bures, Frantisek Plasil, Ilias Gerostathopoulos, Petr Hnetynka, and Nicklas Hoch. 2013. "Design of Ensemble-Based Component Systems by Invariant Refinement." In *Proc. of CBSE'13*, 91–100. ACM. doi:10.1145/2465449.2465457.

Kramer, Jeff, and Jeff Magee. 2007. "Self-Managed Systems: An Architectural Challenge." In *Proc. of FOSE'07*, 259–68. IEEE. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4221625.

———. 2009. "A Rigorous Architectural Approach to Adaptive Software Engineering." *Journal of Computer Science and Technology* 24 (2): 183–88. doi:10.1007/s11390-009-9216-5.

Lamsweerde, Axel Van. 2008. "Requirements Engineering: From Craft to Discipline." In *Proc. of FSE '08*, 238–49. ACM. doi:10.1145/1453101.1453133.

Morandini, Mirko, and Anna Perini. 2008. "Towards Goal-Oriented Development of Self-Adaptive Systems." In *Proc. of SEAMS '08*, 9–16. ACM. doi:10.1145/1370018.1370021.

NIST. 2012. "Cyber-Physical Systems: Situation Analysis of Current Trends , Technologies , and Challenges."

Ofria, Charles, and Claus O. Wilke. 2004. "Avida: A Software Platform for Research in Computational Evolutionary Biology." *Artificial Life* 10 (2): 191–229. doi:10.1162/106454604773563612.

Patikirikorala, Tharindu, Alan Colman, Jun Han, and Liuping Wang. 2012. "A Systematic Survey on the Design of Self-Adaptive Software Systems Using Control Engineering Approaches." In *Proc. of SEAMS'12*, 33–42. IEEE. doi:10.1109/SEAMS.2012.6224389.

Perrouin, Gilles, Brice Morin, Franck Chauvel, Franck Fleurey, Jacques Klein, Yves Le Traon, Olivier Barais, and Jean-Marc Jezequel. 2012. "Towards Flexible Evolution of Dynamically Adaptive Systems." In *Proc. of ICSE '12*, 1353–56. IEEE. http://dl.acm.org/citation.cfm?id=2337223.2337416.

Ramirez, Andres J., and Betty H. C. Cheng. 2010. "Design Patterns for Developing Dynamically Adaptive Systems." In *Proc. of SEAMS '10*, 49–58. SEAMS '10. New York, NY, USA: ACM. doi:10.1145/1808984.1808990.

Ramirez, Andres J., Betty HC Cheng, Philip K. McKinley, and Benjamin E. Beckmann. 2010. "Automatically Generating Adaptive Logic to Balance Non-Functional Tradeoffs during Reconfiguration." In *Proceedings of the 7th International Conference on Autonomic Computing*, 225–234. ACM. http://dl.acm.org/citation.cfm?id=1809080.

Ramirez, Andres J., David B. Knoester, Betty HC Cheng, and Philip K. McKinley. 2009. "Applying Genetic Algorithms to Decision Making in Autonomic Computing Systems." In *Proc. of ICAC '09*, 97–106. ACM. doi:10.1145/1555228.1555258.

Salehie, Mazeiar, and Ladan Tahvildari. 2009. "Self-Adaptive Software: Landscape and Research Challenges." *ACM Transactions on Autonomous and Adaptive Systems* 4 (2, May): 1–40. doi:10.1145/1516533.1516538.

———. 2012. "Towards a Goal-Driven Approach to Action Selection in Self-Adaptive Software." *Softw. Pract. Exper.* 42 (2): 211–233. doi:10.1002/spe.1066.

Souza, Vítor E. Silva, Alexei Lapouchnian, Konstantinos Angelopoulos, and John Mylopoulos. 2012. "Requirements-Driven Software Evolution." *Computer Science - Research and Development* 28 (4): 311–29. doi:10.1007/s00450-012-0232-2.

Souza, Vítor E. Silva, Alexei Lapouchnian, and John Mylopoulos. 2012. "(Requirement) Evolution Requirements for Adaptive Systems." In *Proc. of SEAMS '12*, 155–64. doi:10.1109/SEAMS.2012.6224402.

Sykes, Daniel, William Heaven, Jeff Magee, and Jeff Kramer. 2008. "From Goals to Components: A Combined Approach to Self-Management." In *Proc. of SEAMS '08*, 1–8. ACM. doi:10.1145/1370018.1370020.

Whittle, Jon, Pete Sawyer, Nelly Bencomo, Betty H. C. Cheng, and Jean-Michel Bruel. 2010. "RELAX: A Language to Address Uncertainty in Self-Adaptive Systems Requirement." *Requirements Engineering* 15 (2): 177–96. doi:10.1007/s00766-010-0101-0.

Yuan, Eric, Naeem Esfahani, and Sam Malek. 2014. "Automated Mining of Software Component Interactions for Self-Adaptation." In *Proc. of SEAMS '14*, 27–36. ACM. doi:10.1145/2593929.2593934.

Zhang, Ji, and Betty H. C. Cheng. 2006. "Model-Based Development of Dynamically Adaptive Software." In *Proc. of ICSE '06*, 371–380. ICSE '06. New York, NY, USA: ACM. doi:10.1145/1134285.1134337.