Rango: An Intuitive Rule Language for Learning Classifier Systems in Cyber-Physical Systems

Melanie Feist[*], Martin Breitbach[†], Heiko Trötsch[†], Christian Becker[‡], Christian Krupitzer[§]
[*]*Johann Wolfgang Goethe University Frankfurt am Main, Germany*
Email: feist@em.uni-frankfurt.de
[†]*University of Mannheim, Germany*
Email: {martin.breitbach, heiko.troetsch}@uni-mannheim.de
[‡]*University of Stuttgart, Germany*
Email: christian.becker@ipvs.uni-stuttgart.de
[§]*University of Hohenheim, Germany*
Email: christian.krupitzer@uni-hohenheim.de

***Abstract*—Self-adaptation is crucial for cyber-physical systems (CPS) to meet their requirements in environments characterized by complexity and uncertainty. As many situations that CPS encounter at runtime are not foreseeable at design time, (online) learning approaches are attractive for such systems. Learning classifier systems (LCS) are a promising learning approach for CPS thanks to their rather low computational complexity. They operate on a set of rules that describe potential adaptation behavior. So far, specifying rules for a learning classifier system is a tedious task that requires expert knowledge. In this paper, we present *Rango* — an intuitive rule language for learning classifier systems — to overcome this challenge. Compared to existing approaches, *Rango* has a strong focus on CPS and provides a large variety of corresponding keywords. In addition, *Rango* rules are automatically transferred into a representation that is usable in a learning classifier system without any modifications. *Rango* therefore empowers system administrators to formulate rules and, hence, leverage an online learning approach for their use case without having prior experience with learning classifier systems. We evaluate *Rango* extensively with (i) a complexity analysis of parsing and rule evaluation, (ii) a usefulness study which shows that *Rango* facilitates both the writing of rules and the understanding of LCS output and (iii) a usability study, which proves that basic programming knowledge is sufficient to understand and formulate *Rango* rules.**

***Index Terms*—learning classifier systems, adaptation, cyber-physical systems, rule language, context-free grammar**

I. INTRODUCTION

In recent years, cyber-physical systems (CPS) found their path into areas such as automotive systems, remote patient monitoring, or smart manufacturing. CPS are complex to develop and operate as they typically act in environments that involve uncertainty; hence CPS require robustness to failures and threats [1], so self-adaptation [2] is essential for CPS to cope with ever-changing environments [3].

In CPS, learning, i.e., improving the adaptation behavior based on past experience, is especially important for two reasons. First, environmental uncertainty leads to situations during runtime that were not predictable at design time [4]; consequently, this increases the complexity for system development as the system has to find ways, i.e., adaptations, to cope with the changes. Second, the effectiveness of certain adaptive behavior may strongly depend on the current situation [5]. Avoiding an obstacle by quickly changing the lane, for instance, is only effective for an autonomous car if the road conditions and the traffic allow such a maneuver.

While a wide variety of (online) learning techniques exist [6], many of them are rather inapplicable in the CPS domain due to the following reasons. As CPS often rely on processors with comparably low computing power (e.g., electronic control units (ECUs) in cars), computationally intensive learning approaches are not feasible. Furthermore, learning in a CPS use case is often particularly complex due to the inherent uncertainty and the large variety of interacting components in such systems. In addition, many use cases require real-time decision making. Due to these requirements, learning classifier systems (LCS) [7]–[9] are frequently used for learning in CPS (e.g., in [10]–[13]). LCS operate on a set of rules that describe possible adaptations. Based on the reward, i.e., the effectiveness of an action that was observed after applying a certain rule, LCS learn from past experience and identify suitable rules for future adaptations.

Although LCS are well-applicable in the CPS domain, an important challenge remains: The rule set must be formulated. Usually, LCS use bit strings to codify rules [8], [9]. As an overly simplistic example, "11$\rightarrow$1" is a possible representation of "`IF` robotFacesObstacle `AND` robotInMotion `THEN` brake". Analogously, another rule would be "01$\rightarrow$0", which would be equal to "`IF` !robotFacesObstacle `AND` robotInMotion `THEN` !brake". Using such bit strings for rule representation in CPS is, however, not advisable in practice for two reasons. First, the complexity of typical use cases would lead to lengthy bit strings that are unreadable and — more importantly — impossible to write by humans. Second, such bit strings cannot express certain conditions and actions that require, e.g., numerical context variables.

To overcome those practical issues, we introduce *Rango*— a generic and flexible rule language — in this paper. *Rango* enables system administrators to systematically and formally create rules for LCS in an intuitive way. It offers maximum flexibility to define abstract and generic rules independent of a specific use case as well as application-specific rules, if these are desired. *Rango* has two major advantages for specifying adaptation behavior in comparison to existing languages such as *Stitch* [14] and *Ctrl-F* [15]. First, it includes a large set of pre-defined CPS keywords such as `Importance`, which models whether a certain application is safety-critical or not. These CPS-specific keywords are re-usable for many CPS use cases. They free the system administrator from a considerable part of the overall coding effort. The keywords additionally already model *mixed-criticality*, i.e., the co-existence of safety-critical and less critical tasks in one system, which is crucial in CPS like autonomous cars [16]. Second, *Rango* is well-integrated with the learning component. The rules specified in *Rango* are automatically transferred into a binary representation that is usable by a LCS without any modifications. As *Rango* allows system administrators to easily export rules that were learned by the CPS during runtime into a human-readable format, it furthermore paves the way towards explainable artificial intelligence (XAI) [17], [18].

We evaluate *Rango* extensively in a threefold study. First, we perform a complexity analysis of parsing and rule evaluation to determine *Rango*'s overhead. Second, we conduct a usefulness study, which shows that *Rango* facilitates both writing rules and understanding LCS output. Third, we conduct a comprehensive usability study, which shows that even non-experts are easily able to understand and create rules in *Rango*. In the remainder of this paper, we first introduce the fundamentals of self-adaptation with LCS (Section II). Afterwards, we review related work (Section III), introduce *Rango* in all detail (Section IV), evaluate *Rango* (Section V), and summarize the results in a conclusion (Section VI).

II. Self-Adaptation with LCS

Self-adaptation is crucial for mixed-critical CPS to cope with the inherent complexity of such systems. The MAPE-K feedback loop [19] is a suitable architecture to implement the desired self-adaptation capabilities in CPS. It consists of components for (i) **M**onitoring system and environment, (ii) **A**nalyzing monitoring data for required adaptations, (iii) **P**lanning adaptations, and (iv) **E**xecuting them. In addition, the four components share a **K**nowledge base.

The analyzing and planning component of the feedback loop can contain a LCS. LCS-based approaches are popular in organic computing (cf. [11], [20]) and they have been applied several times in the CPS domain [10]–[13]. In comparison to other online learning approaches, LCS are less computationally complex. Therefore, training can be performed on devices with low computational power and decisions can be made in real time, which are both essential requirements in the CPS domain. LCS use a set of *rules* that represent potential adaptations. Based on the *reward*, i.e., the effectiveness of an action, that was observed after applying a certain rule, LCS learn and select suitable rules for future adaptations.

Figure 1 shows a typical LCS-based feedback loop. The monitor receives context information about the state of the mixed-critical CPS, pre-processes it, and forwards relevant

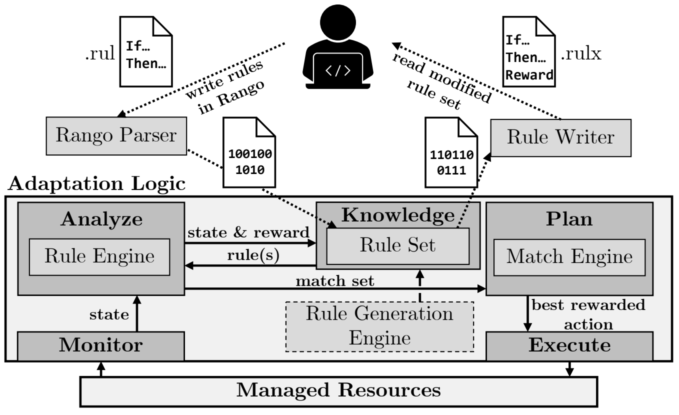

Fig. 1. Typical architecture of an LCS-based feedback loop. The analyzer's rule engine adds rules that are applicable in the current context to the match set. The planner's match engine selects the most promising rule for execution. A rule generation engine may evolve rules at runtime. We propose *Rango*, which is a language that allows system administrators to formulate the rule set in a human-readable format. These rules are parsed and automatically integrated into the learning component. *Rango* also offers a rule writer, which exports rules from the learning component into a human-readable format.

information to the analyzer. The analyzer contains the *rule engine*, which is responsible for rule evaluation and reward calculation. It compares the current system condition to the condition clauses of the *rule set* stored in the knowledge component. All rules that are currently applicable, i.e., all rules where all conditions evaluate to true, are included in the *match set*. The analyzer additionally updates the rule set with the reward that was calculated by it. The planner selects the rule from the match set that is expected to lead to the highest reward. Finally, the executor controls the application of the rules. An LCS-based feedback loop may additionally contain a rule generation engine, which evolves the rule set automatically at runtime, e.g., with genetic algorithms [8], [9].

While this fundamental architecture is well-established, an important challenge remains: The rule set must be formulated. Usually, LCS use bit strings to codify rules [8], [9]. For instance, "11→1" could be a representation of "`IF` robotFacesObstacle `AND` robotInMotion `THEN` brake". Such bit strings are, however, not suitable for rule representation as (i) they are cumbersome to read and write and (ii) they cannot express certain conditions and actions that, e.g., include numerical context variables. To overcome these shortcomings, we introduce *Rango* — a generic and flexible rule syntax — in this paper. As depicted in Figure 1, system administrators store rules written in *Rango* in a `.rul` file. This file is automatically parsed into an internal binary representation that can be used as a rule set of the feedback loop without any modifications. In addition, our approach improves traceability and explainability of learning as it offers the option to export a rule set modified by learning to a `.rulx` file in *Rango*'s human-readable form.

III. Related Work

In an insightful overview [6], D'Angelo *et al.* compare which learning techniques are applied in adaptive systems.

Approaches that rely on Reinforcement Learning (RL) are dominantly applied to realize simple learning tasks (e.g., with Q-Learning) as well as sophisticated ones (e.g., with LCS).

Learning classifier systems. In this paper, we focus on LCS because those and variants — such as the *Extended Classifier System* (XCS) [21] by Wilson [22] or *XCS for real-valued input spaces* (XCSR) [23] — have been widely used for implementing adaptive behavior with runtime learning capabilities in various domains. For instance, they have been applied in typical CPS use cases such as self-adaptive traffic management [10], [11], autonomous parameter adjustment of data communication protocols [12], or Industry 4.0 [13]. Beyond the domain of CPS, Rosenbauer *et al.* [24] applied *XCS for function approximation* (XCFS) for automated test case prioritization and Stein *et al.* targeted a smart cameras application [25]. However, to the best of the authors' knowledge, none of the approaches focuses on supporting the system administrator in the process of the rule set generation. Still, the applied learning principles can be combined with our approach to learn new rules or optimize existing ones.

Adaptation languages. Several requirement modelling languages and modelling approaches such as *FLAGS* [26], *CARE* [27], *RELAX* [28], *LOREM* [29] or [30] exist. The main purpose of such languages is the definition of requirements at design time. At runtime, the resulting models can be used as a knowledge base to support the reasoning for adaptation. Similarly, languages such as *Stitch* [14] and *Ctrl-F* [15] focus on the definition of adaptation behavior. In a previous work, we use the structural specification language *Clafer* [31] to model adaptation behavior and *UML* to model the target system in our *REACT* framework [32] and its extension *REACT-ION* [33]. In another previous work [34], we focus on a clear separation of the adaptation decision logic and the rules. We present an approach where an initial rule set can be defined in a spreadsheet. In comparison to these approaches for specifying adaptation behavior, *Rango* has two key strengths. First, *Rango* is specifically designed for the usage in CPS. It includes, for instance, a large variety of CPS-specific keywords. Second, *Rango* is well-integrated with a learning mechanism. Related languages do not integrate mechanisms to learn or adjust the models, i.e., for representing new adaptation rules.

Reasoning of adaptation under uncertainty. Another research stream focuses on the reasoning of adaptation under uncertainty [35]–[39], which is typical for CPS. *SimCA** [40] provides a control-theoretic approach that offers guarantees for uncertainty related to system parameters, component interactions, system requirements, and environmental uncertainty. Those works do not focus on expressing the uncertainty neither as part of a modelling/programming language nor as rules but instead apply statistical approaches such as Markov decision processes or Bayesian optimization to reflect uncertainty. In addition, those approaches do not integrate learning.

In summary, this paper contributes to the state of the art by introducing the rule language *Rango*. *Rango* builds upon existing research on LCS and offers an intuitive way to specify an initial rule set, which was cumbersome for system administrators so far. In contrast to prominent languages for adaptation behavior such as *Stitch* or *Ctrl-F*, *Rango* is characterized by a strong focus on CPS and the automatic integration of learning.

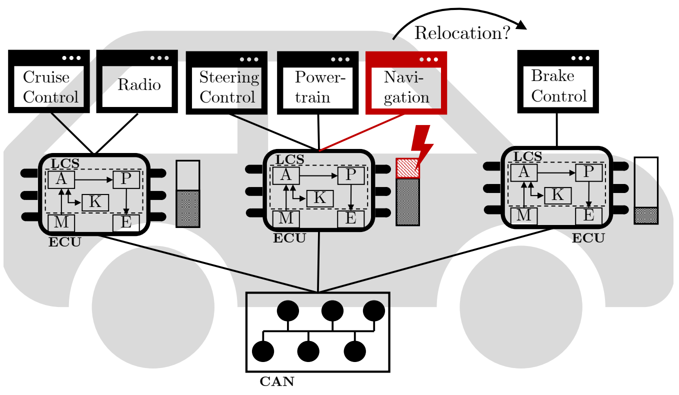

Fig. 2. Running example from the automotive domain. A car contains several ECUs that execute safety-critical and non-critical applications. After the driver activates the navigation system, the corresponding ECU has to migrate an application to another ECU due to a lack of processing power. We show how *Rango* can be used to formulate this desired adaptation behavior.

IV. Rango — A Rule Language for LCS-Based Adaptation in CPS

In this section, we describe *Rango*'s features by applying it to a simplified automotive use case. After presenting the use case, we explain *Rango*'s syntax, its CPS-specific features, its rule files, and how those rules are evaluated.

A. Running Example

Throughout the remainder of this paper, we refer to a running example from the automotive domain. The system model implied by this CPS use case is shown in Figure 2. A car has several electronic control units (ECUs), i.e., processor *nodes* with a certain computational power. Each ECU can be responsible for various car functions such as brake control, cruise control, or steering. Apart from those safety-related functions, the ECUs also control comfort and assistance functions including navigation or infotainment. We refer to these functions as *applications* in the following. Each application has predefined requirements such as timing constraints which should be complied to at all time. The communication among ECUs is realized via a typical vehicular *communication channel* (e.g., CAN, time-triggered ethernet, or FlexRax).

The degree to which a component (i.e., a node, an application, or a communication channel) meets its requirements is called *health*. The health value is in the range of $-infinity$ to 1, where 1 means completely healthy and all values less than or equal to 0 mean unhealthy. This abstraction makes a flexible mapping of different requirements to a single numeric value possible. For instance, if an application is starting to miss deadlines more frequently, the application's health value declines. Similarly, a node may become unhealthy, e.g., if the processor load exceeds 80%. In this case, the node might not be able to cope with the computational demand of its applications in peak times.

During the journey, the number of active and required applications in the car varies. For instance, the driver may start the navigation system due to a road closure. This potentially leads to an ECU overload if the ECU on which the navigation application is started does not have sufficient computational power to cope with the increased demand. As a consequence, the applications on this ECU may fail to meet their requirements. Thus, the local ECU and the corresponding applications become unhealthy. The overall goal of self-adaptation in this use case is to maintain a good health value for each component. To achieve this, each ECU has its own LCS-based feedback loop. If the communication channel within the system is still healthy, a relocation of the unhealthy application to a healthy ECU is a suitable adaptation in the above scenario. In the following, we show how we can specify this adaptation behavior easily with *Rango*.

B. Rango Syntax

The complete *Rango* grammar describing the syntax of the language in Extended Backus-Naur Form (EBNF) is available on GitHub (https://github.com/MelFeist/Rango). The adaptation behavior that is desirable in the example use case — a relocation of an unhealthy application — can be specified with the following rule in *Rango*:

```
If relocationMightBeUseful
Then relocateLocalUnhealthyApp
```

In general, a rule consists of a *condition* and an *action*. Conditions and actions can either be specified directly or defined and referenced by name as shown in the *Rango* code snippet above. Here, the name of the condition is `relocationMightBeUseful` and the action's name is `relocateLocalUnhealthyApp`. The reference by name allows system administrators to use an action or condition multiple times in a rule set without having to rewrite it. In addition, this increases the efficiency of rule evaluation within the feedback loop, since a condition/action that is used multiple times only needs to be evaluated once.

We need to define a condition or action to be able to reference it by name. We first define the `relocationMightBeUseful` condition. This condition checks whether unhealthy applications are running on the local ECU while (i) the communication is still healthy and (ii) there are other suitable ECUs in the car to which the unhealthy applications can be migrated. In *Rango*, we formulate this condition as follows:

```
DefineCondition relocationMightBeUseful :
Cardinal (localUnhealthyApps) > 0 And
Cardinal (localUnhealthyNodes) > 0 And
Cardinal (nonLocalSuitableNodes) > 0 And
Cardinal (localUnhealthyComms) = 0
```

This example shows that a condition consists of conjunctive links (`And`) of *sub-conditions*. Sub-conditions always operate on *sets*. A set can contain applications, nodes, or communication channels. The `Cardinal` keyword delivers the cardinality of a set. In addition to their cardinality, the maximum, minimum, or average value of one of the set's attributes can be used for comparison. In the code snippet above, the cardinality of four sets (`localUnhealthyApps`, `localUnhealthyNodes`, `nonLocalSuitableNodes`, `localUnhealthyComms`) is used to specify sub-conditions. The overall condition `relocationMightBeUseful` therefore evaluates to true if the first three sets are non-empty while the last one is empty.

Rango offers two options for the definition of sets. First, sets can be defined by directly referencing applications, nodes, or communication channels by name. Second, sets can be defined via *queries*, e.g., by selecting all applications with a health value smaller than 0. Defining sets via queries makes it possible to keep the rules independent of specific applications. Instead, *Rango* constructs these sets dynamically based on the respective query. In a query, the attributes of a node, application, or communication channel can be compared to arbitrary values or their maximum/minimum value can be requested. Two exemplary set definitions with queries are:

```
DefineSet localUnhealthyApps :
App Local Where Health <= 0

DefineSet nonLocalSuitableNodes :
Node NonLocal Where
Capacity >= Demand (localUnhealthyApps),
Health Max
```

Here, the first set (`localUnhealthyApps`) includes all local unhealthy applications, i.e., those with health values of 0 and below. The second set (`nonLocalSuitableNodes`) contains the most healthy non-local nodes with sufficient remaining computing power[1]. Similar to conditions and actions, a set can be used several times by referencing to its name. A set that is used multiple times is only constructed once per rule evaluation period.

So far, we have introduced all of *Rango*'s features that are required to specify the condition. Now, we proceed with the definition of the action (`relocateLocalUnhealtyApp`):

```
DefineAction relocatelocalUnhealthyApp :
Relocate localUnhealthyApps
To nonLocalSuitableNodes
```

We observe that this action definition re-uses the sets that we have already defined above. An action can be applied either to all elements of the set or to one element. This is realized by either using or omitting the `All` keyword. Without the `All` keyword, only the first element of the `localUnhealthyApps` set is moved to the first element of the `nonLocalSuitableNodes` set. If we use the `All` keyword before `localUnhealthyApps` (`Relocate All localUnhealthyApps`...), all

[1]If there is more than one element in the `localUnhealthyApps` set, the demand of the first application of this set is used for comparison. It is also possible to use the demand of all applications in the set with the `All` keyword.

elements of the application set will be moved to the first element of the `nonLocalSuitableNodes` set. If we use the `All` keyword before both sets (... `To All nonLocalSuitableNodes`), all elements of the application set will be moved to the elements of the node set according to the following scheme: the i-th element of the application set is migrated to the ($i \bmod |nodeset|$)-th element of the node set.

C. CPS-Specific Elements in Rango

While it is possible to apply *Rango* in arbitrary use cases, the language includes four features that are particularly useful in CPS development: CPS-specific *components*, *attributes*, *adaptation actions*, and *scopes* (cf. Table I[2]). We demonstrate each of these features with the following code snippet:

```
DefineSet leastImportantApps:
App System Where Importance Min

If systemOverloaded
Then Stop leastImportantApps
```

This code snippet specifies a rule that stops the least critical application in the CPS in case of an overload[3].

We already introduced queries in the previous section to group several components dynamically into sets. To achieve the desired behavior, we must first define a set that contains the least critical application(s). Therefore, *Rango* includes keywords that automatically address typical components of a CPS (applications, nodes, and communication channels). In the code snippet, we use the keyword `App` to consider all applications. Furthermore, attributes of the components such as their health are used for comparison or maximum/minimum determination. *Rango* offers system administrators 43 pre-defined CPS-specific attributes that are ready to use.

In the above code snippet, we use *Rango*'s CPS-specific attribute "importance" (i.e., the keyword `Importance`). This keyword allows system administrators to consider mixed-criticality by making it possible to specify rules that realize a dynamic prioritization of critical tasks, e.g., in overload situations as in the above code snippet. In the provided example the prioritization is achieved by terminating (`Stop`) the least important and thus least critical application in the entire system. This is only one way to solve the problem. There might be other, even better, adaptation actions to resolve the overload. Since we strive for a maximum of flexibility and opportunities for adaptation to be able to learn the best actions in different situations, *Rango* includes 18 keywords that represent adaptation actions in CPS such as migrating applications (`Relocate`).

Finally, we introduce different scopes in *Rango*. In practice, multiple CPS may cooperate and interact with each other over a network to achieve their goals. Such a network of CPS is often referred to as a cyber-physical network (CPN). For example, in a platooning [41] scenario, in which several autonomous vehicles drive in a convoy with small inter-vehicle distances, those vehicles need to coordinate their inter-vehicle gaps via IEEE 802.11p communication. The overarching CPN would therefore consist of multiple CPS (the cars). Each CPS itself includes several processing nodes (i.e., ECUs), which each have their own feedback loop for decision making. In such systems-of-systems, scopes such as "in the current CPS", "on the current processing node", or "somewhere in the whole CPN" play an important role in how system administrators would intuitively describe the adaptation behavior that they wish to model. For instance, a rule may migrate all applications that are running on an ECU to another ECU in the same car and not to an arbitrary ECU in the whole CPN. *Rango* offers multiple scopes with corresponding keywords for the specification of such rules: (i) *Global* refers to all components in the entire CPN (e.g., all applications within the whole platooning scenario), (ii) *System* refers to all components of the corresponding CPS (e.g., all applications running in a single car) (iii) *Local* refers to all components on the same node as the feedback loop (e.g., all applications running on the

TABLE I
CPS SPECIFIC COMPONENTS, ATTRIBUTES, ADAPTATION ACTIONS AND SCOPES INCLUDED IN RANGO

CPS-specific element	**Keyword**	**Description**
Components	`App`	Applications such as brake control
	`Node`	Processing nodes such as ECUs
	`Comm`	Communication channels such as CAN
Attributes	`Importance`	Describes in $[0,\infty)$ the criticality of an application
	`Health`	Describes in $(-\infty,1]$ whether a component meets its requirements
	`Period`	Period in which an application is executed
	`Scheduling`	Scheduling policy of a node or communication channel
	`Capacity`	Remaining computational power of a node or communication bandwidth of a communication channel
	...	38 further CPS-specific attributes
Actions	`Stop`	Terminates an application
	`Relocate`	Migrates an application to another node
	`TunePeriod`	Changes the period of an application
	`SetPriority`	Sets the priority of an application
	`SetScheduling`	Changes the scheduling policy of a node or communication channel
	...	13 further CPS-specific actions
Scopes	`Local`	All components local to a certain node
	`System`	All components in a certain CPS
	`Global`	All components in the entire CPN
	`NonLocal`	All components not local to a certain node
	`NonSystem`	All components not belonging to a certain CPS
	...	4 further scopes

[2]Please refer to GitHub (https://github.com/MelFeist/Rango) for the entire list of available CPS specifics.

[3]For readability reasons, we omit the definition of the `systemOverloaded` condition in the code snippet.

```
1  /* ConfigurationDirectives */
   EvaluationPeriod 0.1
   ---------------------------------------------------------------------------
2  /* Rule Set */
   /* Named Sets */
   DefineSet localUnhealtyApps: App Local Where Health <= 0
   DefineSet leastImportantApps: App Global Where Importance Min

   /* Named Conditions */
   DefineCondition localUnhealtyAppsOnNode:
   Cardinal (localUnhealthyApps) >= 0

   /* Named Actions */
   DefineAction stopLeastImportantApp:
   StopApp leastImportantApps

   /* Rules */
   If localUnhealtyAppsOnNode Then stopLeastImportantApp
```

Fig. 3. Rule file structure .rul

same ECU). Based on these basic scopes, several composite scopes for the rules can be derived, e.g., *NonLocal* refers to all components that are *not* located on the local node, and *NonSystem* refers to all components *not* belonging to the CPS of the local nodes. For example, a component like an application is *Local* to an ECU if the functionality is executed on this ECU. All applications not running on this node can be referred to as *NonLocal* and all applications not belonging to the CPS can be referenced by *NonSystem*.

D. Structure of Rango Rule Files

System administrators store *Rango* rules in `.rul` files. This way, system administrators or domain experts can write the initial rule set or updates of rules without knowledge on LCS or implementation of CPS. As depicted in Figure 3, such files consist of two parts: the configuration part and the rule set. In the configuration part, various configuration directives can be defined. System administrators can specify settings for the rule evaluation times (e.g. `EvaluationPeriod`), action execution, learning parameters, and reward calculation parameters. The second part of the file — the rule set — consists of all sets, conditions, actions, and rules. Any sequence of sets, conditions, actions, and rules is valid as long as there are no forward references (single pass rule parser).

Furthermore, our approach improves traceability and explainability of learning as it allows to export rule sets into a human-readable format after the LCS performed online learning. Such modified rule sets are stored in `.rulx` files. Files in this format starts with a comment that states which LCS modified the rule set. The remainder is structured in the same way as a `.rul` file (cf. Figure 3). However, each rule is now followed by the corresponding `Reward` and the `Experience`. The reward is a real number that expresses the benefit of the rule execution for the system goal and the experience value indicates how often a rule was executed. Such a modified rule file in the `.rulx` format can also be re-parsed in the internal representation and, hence, be re-used as an initial rule set for adaptation. This makes it possible to benefit from previous learning, e.g., performed by another LCS in a similar environment.

E. Rule Evaluation

Rules can basically be evaluated either time or event-driven. In the event-driven evaluation, the rule evaluation takes place exactly when an attribute value of a component (i.e. health, demand, ...) changes. In the case of time-driven evaluation, the rule set is evaluated periodically at fixed time intervals. In our implementation the rules are evaluated time-driven, because firstly this fits perfectly the periodic evaluation structure of LCS and secondly the evaluation period is usually lower than the change frequency of a large number of attributes. Therefore, all rules are evaluated periodically with a definable period p and the associated conditions and actions are determined.

V. Evaluation

We evaluate *Rango* in three experiments. First, we assess the memory and computational overhead of *Rango* for rule sets of different sizes (Section V-A). Second, we consult five LCS experts and let them evaluate the usefulness of *Rango*, i.e., whether *Rango* has a practical worth (Section V-B). Third, we evaluate *Rango*'s usability, i.e., whether it is easy to understand and writes rules using *Rango* without extensive training, in a study with 37 participants, mostly without experience in LCS (Section V-C). Section V-D discusses threats to validity.

A. Experiment 1: Performance Evaluation

Using *Rango* instead of bit strings to formulate rules introduces an overhead in terms of memory usage and computational complexity. In the first experiment, we evaluate this overhead both theoretically and in real-world measurements. We answer three research questions with this experiment:

RQ1: *How much memory does Rango use?*
RQ2: *How computationally complex is the parsing?*
RQ3: *How computationally complex is the rule evaluation at runtime?*

1) Memory Usage (RQ1): The rule set is parsed into an internal data structure that stores four tables: (i) rules table, (ii) conditions table, (iii) actions table, and (iv) sets table. If an action, condition, or set is referenced by name, the respective element is stored only once in the table and can be referenced from elements in the other tables. Thus, the memory requirement M is linear to the number of rules n_{rules}, conditions $n_{conditions}$, actions $n_{actions}$, and sets n_{sets} of the rule set. The memory required for a condition is linear to the number of its sub-conditions ($\mathcal{O}(n_{condition_{maxSub}})$). Similarly, the memory required for a set is linear to the number of query clauses ($\mathcal{O}(n_{set_{maxClauses}})$) describing the set. The overall memory usage M can therefore be described as:

$$M = \mathcal{O}(n_{rules} + n_{conditions} \cdot n_{condition_{maxSub}} + n_{actions} + n_{sets} \cdot n_{set_{maxClauses}}) \tag{1}$$

In practice, an exemplary rule set with 10 rules and respective 6 conditions with a maximum of 2 sub-conditions and 5 actions using 7 sets with a maximum of 2 query elements needs 1083 bytes in a 32 bit system and 601 bytes in a 16 bit system. We therefore conclude that *Rango* leads to negligible memory overhead that is even feasible for microcontrollers.

2) Parsing complexity (RQ2): The *Rango* parser transfers the rule set written by the system administrator into the aforementioned internal data structure. This leads to a certain delay on system startup. After that, the system works on the binary data structure only. As the syntactic definition of *Rango* bases on a context-free LL(1)-grammar, the parsing complexity is $\mathcal{O}(n)$. We measured the parsing complexity in practice on an Intel 11th Gen Core i7-11700 with $2.5\,\mathrm{GHz}$ and $64\,\mathrm{GB}$ RAM for rule files of different sizes. Even the largest files that contain around 100 rules lead to average parsing times of less than $3\,\mathrm{ms}$. On low-performance systems, an external parser on a more powerful machine could transfer the rule set into the binary representation if parsing becomes too time-consuming. This representation can then be loaded to the target system.

3) Evaluation complexity (RQ3): At runtime, rules have to be evaluated, i.e., it has to be checked whether they apply to the current context. The evaluation of each rule follows three steps: (i) determining the condition and action that belong to the rule, (ii) checking whether conditions are fulfilled, and (iii) calculating each required set. Since condition and action can be referenced directly from a rule in the rule table, the complexity of the first step ($C_{(i)}$) is within $\mathcal{O}(1)$.

To check whether a condition is fulfilled, the given set property (cardinality, average, minimum, or maximum element) must be checked for each set of the condition. With the maximum number of sets $n_{condition_{maxSets}}$ in a condition and the maximum number of members $n_{set_{maxMember}}$ in a set, the worst-case complexity of the second step ($C_{(ii)}$) is:

$$C_{(ii)} = \mathcal{O}(n_{condition_{maxSets}} \cdot n_{set_{maxMember}}) \tag{2}$$

To calculate a set (step 3), we need to determine for each component (applications, nodes, communication channels) if they are a member of the set. *Rango* offers two options for the definition of sets. First, sets can be defined by directly referencing members by name. Second, sets can be defined via queries consisting of several clauses describing the properties of set members. Hence, with $n_{components}$ as the maximum number of components in the system and $n_{sets_{maxClauses}}$ as the maximum number of query clauses of a set, the worst case complexity of the third step ($C_{(iii)}$) is:

$$C_{(iii)} = \mathcal{O}(n_{components} \cdot n_{sets_{maxClauses}}) \tag{3}$$

Assuming that the rule set consists of n_{rules} rules including $n_{conditions}$ conditions and n_{sets} sets, the overall complexity C for the evaluation of all rules is:

$$\begin{aligned} C =& \mathcal{O}(n_{rules} \cdot C_{(i)} + n_{conditions} \cdot C_{(ii)} + n_{sets} \cdot C_{(iii)}) \\ =& \mathcal{O}(n_{rules} + n_{sets} \cdot n_{components} \cdot n_{sets_{maxClauses}} + \\ & n_{conditions} \cdot n_{condition_{maxSets}} \cdot n_{set_{maxMember}}) \end{aligned} \tag{4}$$

If we assume a fixed size rule set, the evaluation complexity scales linearly with the number of components in the CPN and the sets ($\mathcal{O}(n_{components} + n_{sets_{maxMember}})$). If we instead assume a fixed environment and a variable size rule set where the maximum number of clauses in a set and maximum number of sets in a condition is limited, the evaluation complexity scales linear with the number of rules, conditions and sets ($\mathcal{O}(n_{rules} + n_{conditions} + n_{sets})$). We measured the time required in practice for the evaluation of different rule sets in a fixed environment on the aforementioned evaluation PC. The average evaluation time for rule sets of around 100 rules was well below $0.1\,\mathrm{ms}$. Thus, we conclude that the evaluation of *Rango* rules leads to a negligible overhead only.

B. Experiment 2: Usefulness Study

After showing that *Rango* leads to marginal overhead, we now assess its usefulness, i.e., whether *Rango* has a practical worth for its potential users. *Rango* supports them in two ways: (i) they can express an initial rule set for an LCS with the language and (ii) they receive human-readable output after learning that helps to trace and explain the learning process. Thus, we answer two research questions with this experiment:

RQ4: *How useful is Rango for writing an LCS rule set?*
RQ5: *How useful is Rango for understanding and interpreting LCS output data?*

1) Procedure & Methodology: To answer the research questions, we designed an online questionnaire[4] for LCS experts, which consists of three parts. In the first part, the experts had to describe the status quo, e.g., which typical problems they face while using LCS or how they usually formulate rule sets. In addition, they had to rate how difficult it is to define an initial LCS rule set for (i) experts like themselves and (ii) system designers or administrators working in the industry on a 5-point Likert scale from 1 ("very hard") to 5 ("very easy"). Similarly, they had to rate how difficult it is to understand and interpret typical LCS output for both groups on the same scale. In the second part, we introduced *Rango* to the experts with a textual description similar to Section IV of this paper. In the third part, the experts again had to rate how difficult it is to define a rule set for themselves and system administrators but this time assuming that they could use *Rango*. The same applies to understanding and interpreting LCS output. The experts were additionally asked to provide feedback on *Rango* including its syntax, missing features, and — most importantly — its overall usefulness ("Would you use the language if it would be available?").

2) Participants: We applied snowball sampling to acquire participants. We contacted several LCS experts and asked them to recommend further participants. Five LCS experts completed the questionnaire. On a scale from 1 ("not familiar") to 5 ("very familiar"), the self-rated average experience with LCS among the participants is 4.6. Four out of five participants have additionally used LCS to make a system adaptive.

3) Results: In the first part of the questionnaire — before introducing *Rango* — three out of the five LCS experts have mentioned "rule encoding" or "finding good initial rules" as typical problems while working with LCS, which matches the motivation of this paper. While the quality of the rules is in

[4]The whole questionnaire is available for the reviewers at https://forms.gle/fgQK9cfcoqgM3XYYA

TABLE II
AVERAGE EASINESS OF (I) WRITING RULES AND (II) UNDERSTANDING LCS OUTPUT ON A SCALE FROM 1 ("VERY HARD") TO 5 ("VERY EASY"), ONCE WITH AND ONCE WITHOUT RANGO

Task	**Group**	**Status quo**	**With Rango**
Writing rules	LCS Experts	3.2	4.4
	System administrators	2.8	3.4
Understanding output	LCS Experts	3.4	4.0
	System administrators	2.2	3.6

theory independent from its formal representation, we believe that *Rango* can help to find better initial rules as these rules can be expressed easier in a more intuitive format.

Two experts mentioned that they usually formulate the initial rule set in bit strings. Improving this cumbersome way to specify rules is the core idea behind *Rango*. Another two experts start the learning process with an empty rule set and rely on automated, random generation. Consequently, such an approach might lead to rules that neither can be interpreted by humans nor it is traceable how the system found the rules. Especially in CPS use cases, which are typically complex, defining an initial rule set with low effort — as possible with *Rango* — is helpful to accelerate the learning process.

In the questionnaire, the LCS experts rated the easiness of writing a rule set and understanding the LCS output, both with and without *Rango*. They also estimated the easiness if system administrators had to write rules or understand the output. We summarize the results in Table II. These results provide three insights. First, *Rango* facilitates the process of writing an initial rule set and understanding the LCS output for both LCS experts and system administrators. Second, as far as writing rules is concerned, especially LCS experts benefit from *Rango*. The experts rate the easiness of writing rules with *Rango* 4.4 out of 5 (instead of 3.2 without *Rango*). For system administrators, the improvement is smaller (2.8 without *Rango*, 3.4 with *Rango*). One explanation for this observation is that system administrators lack experience with rule-based system and, hence, find it more difficult to write such rules. Still, it can be observed that *Rango* improves the score by around 30%. Third, as far as understanding output is concerned, *Rango* is expected to be especially helpful for system administrators. The easiness of understanding LCS output increases from 2.2 out of 5 to 3.6. In comparison, the improvement for LCS experts is smaller (3.4 without *Rango*, 4.0 with *Rango*). We believe that the improvement was smaller since the output was already understandable for LCS experts anyway. This would also match the experts' assessment in another question that the existing output (i.e., without *Rango*) already makes the learning process understandable. They rated the traceability/explainability of the learning process 4.0 out of 5 on a scale from 1 ("not at all") to 5 ("very detailed").

In summary, we conclude from this experiment that LCS experts perceive *Rango* as useful. The above insights from Table II show that *Rango* facilitates both writing rules (research question RQ4) and understanding LCS output (research question RQ5). In addition, the LCS experts rated *Rango*'s syntax 4.2 out of 5 on a 5-point Likert scale. In the final question whether the experts would use *Rango* if available, *Rango* scored 4.2 out of 5 (scale: 1 ("no, never") to 5 ("yes, regularly")). The experts were additionally asked to suggest further features. The majority of these suggestions are already included in *Rango*'s current version but were not explicitly presented to the LCS experts in the textual description. One expert suggested that *Rango* should allow to set bounds for rule mutations. We agree that constructs that restrict, influence, or guide the learning process of an LCS would be a valuable addition in future versions of *Rango*.

C. Experiment 3: Usability Study

So far, we have shown that (i) *Rango* introduces negligible overhead and that (ii) experts perceive the language as a useful addition. In a final experiment, we now evaluate *Rango*'s usability, i.e., how easily users understand and create rules in *Rango*. In this experiment, we answer two research questions:

RQ6: *How easy is it for potential users to understand rules written in Rango?*
RQ7: *How easy is it for potential users to write rules in Rango?*

1) Procedure: Each participant completed an online questionnaire[5] that consisted of three parts. First, participants were confronted with a textual description of a CPS use case from the health care domain and the corresponding code in *Rango* that would achieve the desired adaptive behavior in the use case. The task of the participants was to understand the code step-by-step and to answer two types of questions: (i) multiple choice questions and (ii) open questions. As far as multiple choice questions are concerned, the participants had, for instance, to choose the correct interpretation of a code snippet in natural language among four alternatives. In the open questions, participants were asked to describe the purpose of a code snippet in their own words. The participants did not receive a briefing about *Rango*'s syntax or the idea behind the language during this part of the study. Analyzing the participant's performance in the first part of the study allows us to answer research question RQ6.

Second, participants were confronted with the automotive use case that we use as a running example throughout this paper (cf. Section IV-A). The task of the participants was to write the rule and its required condition, action, and sets from Section IV-B step-by-step by themselves. The textual guidance led the participants through the coding process (e.g., by stating that they have to define an according set now). It additionally contained explanations of *Rango*'s syntax in a format similar to typical tutorials for programming languages available on the internet. As an example, we show the syntax hint for the definition of a condition in Figure 4. The performance of the participants in this second part of the study was analyzed to answer research question RQ7.

[5]The whole questionnaire is available for the reviewers at https://forms.gle/649bMCth1F78DuPi6

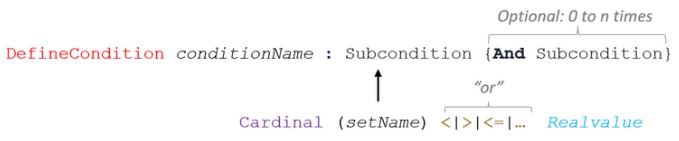
Optional: 0 to n times
DefineCondition conditionName : Subcondition {And Subcondition}
"or"
Cardinal (setName) <|>|<=|... Realvalue

Fig. 4. Syntax hint for the condition definition in the writing part of the usability study questionnaire.

Third, participants completed a self-evaluation related to their skills in three areas: (i) programming, (ii) computer science research related to this paper, and (iii) enterprise information systems. The participants had to rate their expertise with regards to several programming languages, technologies, or concepts such as "Python", "Cyber-physical systems", or "ERP" on a scale from 0 ("never heard of it") to 5 ("expert").

2) Methodology: We labeled the participants' answers to the multiple choice questions with either "correct" or "incorrect". The answers to the open questions with regards to understanding rules were categorized into "correct", "inaccurate", and "incorrect". As far as the second part of the study (rule writing) is concerned, we labeled the code that was provided for each question as either "correct", "with minor syntactic mistake(s)", "with semantic mistake(s)", or "incorrect". To analyze the third part of the study (self-evaluation), we averaged the scores that participants entered for the items in one area to obtain three values between 0 and 5 that describe the proficiency of a participant in each of the three areas (programming, research, and enterprise systems).

3) Participants: The participants were acquired using snowball sampling. In total 37 participants (28 male, 8 female, 1 diverse) took part in the study. Their age ranges between 20 and 63 (average = 29) years. The participants had to give their highest academic degree: 48.6% have a Master's degree, 29.7% a Bachelor's degree, 13.5% a PhD, and 8.1% no university degree. Most of the participants (62.2%) are studying or working in the field of computer science. The participants' programming experience ranges between 0 and 35 (average = 7) years. In the self-evaluation, the participants achieved an average score of 2.5 out of 5 in programming, 2.4 in computer science related to this paper, and 1.9 in the enterprise information systems area.

4) Results: Figure 5 visualizes the results to answer the research questions RQ6 and RQ7. Understanding the code step-by-step and answering multiple choice questions and open question achieved a high percentage of correctness. This leads to the conclusion, that rules written in *Rango* are easy to understand and that no explicit syntax briefing is required (research question RQ6).

In the writing part, the participants were asked to write a rule and its required condition, action and sets. In each part of the task a small number of minor syntax mistake(s) like wrong capitalization and forgotten brackets were made. In practice, the parser would note these errors and the user would get a notification to correct the syntactic mistakes. In the following, we therefore count these answers as correct.

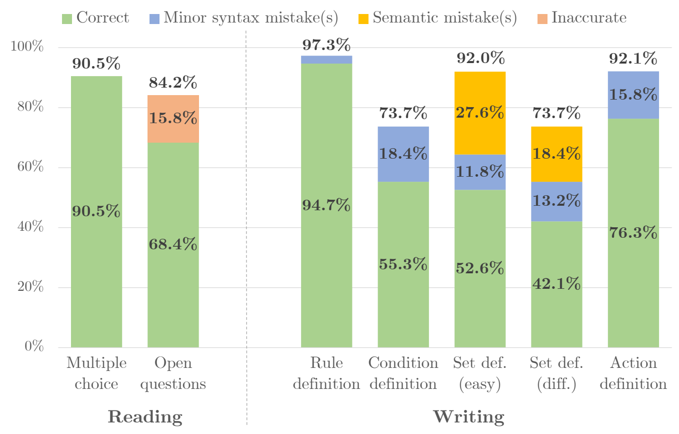

Fig. 5. Results of the rule reading and writing part of the usability study.

Furthermore, only in the definition of sets semantic mistakes were made. One common mistake was the usage of the wrong scope for the set specification. Also, in the definition of the difficult `nonLocalSuitableNodes` set, the specification of the application set was forgotten after the `Demand` attribute. Furthermore, the figure shows that the definition of the condition and sets is more demanding for the users. Still, more than half of the participants (55.3%) correctly defined the difficult set. The easier sets were correctly defined by 64.4% and the condition was formulated correctly by 73.7% of the participants. The definition of the action was correct by 92.1% and nearly all participant defined the rule correctly (97.3%). The analysis of the data shows that even non-experts are easily able to create rules in *Rango* (research question RQ7). We further concluded, that it is useful for the application of *Rango* to explicitly explain the predefined scopes and the application of the predefined attributes in more detail to ease the definition of the conditions and sets. This was not the case in the study.

D. Threats to Validity

Although we have evaluated *Rango*'s overhead, usefulness, and usability extensively, potential threats to validity remain. First, the overhead in terms of memory usage and computational complexity was measured on a desktop PC. While the results strongly suggest that typical microcontrollers used in the CPS domain could cope with the overhead, evaluating *Rango* on various microcontrollers is part of future work. Second, the LCS experts rated *Rango*'s usefulness solely based on a description of the language and its features, not based on their experience with the language in their everyday work. Third, using a questionnaire to assess *Rango*'s usability could potentially introduce a bias compared to a real-world usage of *Rango*, which would include, e.g., debugging and gradually evolving the rule set. Fourth, the majority of the participants in the usability study were non-experts without experience with CPS, LCS, or self-adaptive systems. Therefore, future work might include an extended study where the intended users of *Rango* such as system administrators or LCS experts apply the language to solve typical problems of their everyday work.

VI. Conclusion

This paper introduces *Rango* — a language that allows system administrators to intuitively specify a rule set for LCS. Rules written in *Rango* are automatically transferred into a binary format that is usable by a LCS without any modifications. As *Rango* includes more than 50 CPS-specific keywords, it is especially powerful for adaptation in CPS use cases. In an extensive evaluation, we have shown that *Rango* (i) introduces only marginal memory and computational overhead, (ii) is perceived as useful by LCS experts, and (iii) is intuitive and therefore usable even by non-experts.

As future work, we plan to evaluate *Rango* in an extended real-world study where system administrators use the language to solve problems from their everyday work. Therefore, we will integrate *Rango* with the *Chameleon* middleware [16] to offer a comprehensive approach to self-adaptation in CPS. *Rango* will be used as input for the implemented LCS in the middleware. Thus, the whole management and adaptation capabilities needed to handle the complexity of CPS is encapsulated in a single framework. System administrators can write initial rule sets and easily use them in different application scenarios. Since CPS may necessitate a distributed rule evaluation, the middleware additionally offers strategies for interference treatment and reward calculation.

References

[1] T. Bures et al., “Software Engineering for Smart Cyber-Physical Systems – Towards a Research Agenda,” *ACM SIGSOFT Software Engineering Notes*, vol. 40, no. 6, 2015.

[2] C. Krupitzer, F. M. Roth, S. VanSyckel, G. Schiele, and C. Becker, “A Survey on Engineering Approaches for Self-Adaptive Systems,” *PMCJ*, vol. 17, no. Part B, 2015.

[3] H. Muccini, M. Sharaf, and D. Weyns, “Self-adaptation for cyber-physical systems: A systematic literature review,” in *Proc. SEAMS*. ACM, 2016.

[4] S. Tomforde and C. Müller-Schloer, “Incremental Design of Adaptive Systems,” *J. Ambient Intell. Smart Environ.*, vol. 6, no. 2, 2013.

[5] E. M. Fredericks, I. Gerostathopoulos, C. Krupitzer, and T. Vogel, “Planning as optimization: Dynamically discovering optimal configurations for runtime situations,” in *Proc. SASO*. IEEE, 2019.

[6] M. D’Angelo et al., “On learning in collective self-adaptive systems: state of practice and a 3d framework,” in *Proc. SEAMS*. IEEE, 2019.

[7] J. H. Holland and J. S. Reitman, “Cognitive Systems Based on Adaptive Algorithms,” *Pattern-Directed Inference Systems*, 2009.

[8] O. Sigaud and S. Wilson, “Learning Classifier Systems: A Survey,” *Soft Computing*, vol. 11, no. 11, 2009.

[9] R. J. Urbanowicz and J. H. Moore, “Learning Classifier Systems: A Complete Introduction, Review, and Roadmap,” *Journal of Artificial Evolution and Applications*, 2009.

[10] A. Stein, D. Rauh, S. Tomforde, and J. Hähner, “Interpolation in the eXtended Classifier System: An Architectural Perspective,” *Journal of Systems Architecture*, 2017.

[11] M. Sommer, S. Tomforde, and J. Hähner, “An Organic Computing Approach to Resilient Traffic Management,” in *Autonomic Road Transport Support Systems*, T. L. McCluskey, A. Kotsialos, J. P. Müller, F. Klügl, O. Rana, and R. Schumann, Eds. Springer, 2016.

[12] S. Tomforde and J. Hähner, “Organic Network Control – Turning Standard Protocols Into Evolving Systems,” in *Biologically Inspired Networking and Sensing: Algorithms and Architectures*, P. Lio and D. Verma, Eds. IGI, 2011.

[13] M. Heider, D. Pätzel, and J. Hähner, “Towards a Pittsburgh-Style LCS for Learning Manufacturing Machinery Parametrizations,” in *Proc. GECCO Companion*. ACM, 2020.

[14] S.-W. Cheng and D. Garlan, “Stitch: A language for architecture-based self-adaptation,” *Journal of Systems and Software*, vol. 85, no. 12, 2012.

[15] F. Alvares, E. Rutten, and L. Seinturier, “Behavioural Model-based Control for Autonomic Software Components,” in *Proc. ICAC*. IEEE, 2015.

[16] M. Feist and C. Becker, “A Comprehensive Approach of a Middleware for Adaptive Mixed-Critical Cyber-Physical Networking,” in *Proc. PerCom Workshops*. IEEE, 2022.

[17] H. Hagras, “Toward Human-Understandable, Explainable AI,” *IEEE Computer*, vol. 51, no. 9, 2018.

[18] S. Mohseni, N. Zarei, and E. D. Ragan, “A Multidisciplinary Survey and Framework for Design and Evaluation of Explainable AI Systems,” *ACM Trans. Interact. Intell. Syst.*, vol. 11, no. 3-4, 2021.

[19] J. O. Kephart and D. M. Chess, “The vision of autonomic computing,” *Computer*, vol. 36, no. 1, 2003.

[20] A. Stein, R. Maier, L. Rosenbauer, and J. Hähner, “XCS Classifier System with Experience Replay,” in *Proc. GECCO*. ACM, 2020.

[21] A. Stein and S. Tomforde, “Reflective Learning Classifier Systems for Self-Adaptive and Self-Organising Agents,” in *Proc. ACSOS-C*. IEEE, 2021.

[22] S. W. Wilson, “Classifier Fitness Based on Accuracy,” *Evolutionary Computation*, vol. 3, no. 2, 1995.

[23] A. R. M. Wagner and A. Stein, “On the Effects of Absumption for XCS with Continuous-Valued Inputs,” in *Applications of Evolutionary Computation*, P. A. Castillo and J. L. Jiménez Laredo, Eds. Springer, 2021.

[24] L. Rosenbauer, D. Pätzel, A. Stein, and J. Hähner, “Transfer Learning for Automated Test Case Prioritization Using XCSF,” in *Applications of Evolutionary Computation*, P. A. Castillo and J. L. Jiménez Laredo, Eds. Springer, 2021.

[25] A. Stein, S. Rudolph, S. Tomforde, and J. Hähner, “Self-learning Smart Cameras - Harnessing the Generalization Capability of XCS,” in *Proc. IJCCI*. SCITEPRESS, 2017.

[26] L. Baresi, L. Pasquale, and P. Spoletini, “Fuzzy Goals for Requirements-Driven Adaptation,” in *Proc. RE*. IEEE, 2010.

[27] N. A. Qureshi, I. J. Jureta, and A. Perini, “Towards a Requirements Modeling Language for Self-Adaptive Systems,” in *Requirements Engineering: Foundation for Software Quality*. Springer, 2012.

[28] J. Whittle, P. Sawyer, N. Bencomo, B. H. C. Cheng, and J.-M. Bruel, “RELAX: a language to address uncertainty in self-adaptive systems requirement,” *Requirements Engineering*, vol. 15, no. 2, 2010.

[29] D. M. Berry, B. H. C. Cheng, and J. Zhang, “The Four Levels of Requirements Engineering for and in Dynamic Adaptive Systems,” in *Proc. REFSQ*, 2005.

[30] A. J. Ramirez, B. H. C. Cheng, N. Bencomo, and P. Sawyer, “Relaxing Claims: Coping with Uncertainty While Evaluating Assumptions at Run Time,” in *Model Driven Engineering Languages and Systems*. Springer, 2012.

[31] K. Bak, Z. Diskin, M. Antkiewicz, K. Czarnecki, and A. Wasowski, “Clafer: unifying class and feature modeling,” *Softw. Syst. Model.*, vol. 15, no. 3, 2016.

[32] M. Pfannemüller et al., “REACT: A Model-Based Runtime Environment for Adapting Communication Systems,” in *Proc. ACSOS*. IEEE, 2020.

[33] ——, “REACT-ION: A Model-based Runtime Environment for Situation-aware Adaptations,” *ACM TAAS*, vol. 14, no. 4, 2020.

[34] C. Krupitzer et al., “Using Spreadsheet-defined Rules for Reasoning in Self-Adaptive Systems,” in *Proc. PerCom Workshops*. IEEE, 2018.

[35] N. Esfahani, E. Kouroshfar, and S. Malek, “Taming Uncertainty in Self-adaptive Software,” in *Proc. ESEC/FSE*. ACM, 2011.

[36] G. A. Moreno, J. Cámara, D. Garlan, and B. Schmerl, “Efficient Decision-Making under Uncertainty for Proactive Self-Adaptation,” in *Proc. ICAC*. IEEE, 2016.

[37] G. A. Moreno, J. Cámara, D. Garlan, and M. Klein, “Uncertainty Reduction in Self-Adaptive Systems,” in *Proc. SEAMS*. ACM, 2018.

[38] I. Gerostathopoulos, C. Prehofer, and T. Bures, “Adapting a System with Noisy Outputs with Statistical Guarantees,” in *Proc. SEAMS*. ACM, 2018.

[39] C. Kinneer, Z. Coker, J. Wang, D. Garlan, and C. L. Goues, “Managing Uncertainty in Self-Adaptive Systems with Plan Reuse and Stochastic Search,” in *Proc. SEAMS*. ACM, 2018.

[40] S. Shevtsov, D. Weyns, and M. Maggio, “SimCA*: A Control-Theoretic Approach to Handle Uncertainty in Self-Adaptive Systems with Guarantees,” *ACM TAAS*, vol. 13, no. 4, 2019.

[41] V. Lesch, M. Breitbach, M. Segata, C. Becker, S. Kounev, and C. Krupitzer, “An overview on approaches for coordination of platoons,” *IEEE Transactions on Intelligent Transportation Systems*, 2021.