



# Optimal Reconfiguration of Dynamic Software Product Lines Based on Performance-Influence Models

Markus Weckesser

Roland Kluge

TU Darmstadt

markus.weckesser@es.tu-darmstadt.de

roland.kluge@es.tu-darmstadt.de

Michael Matthé

Andy Schürr

TU Darmstadt

michael.matthe@stud.tu-darmstadt.de

andy.schuerr@es.tu-darmstadt.de

Martin Pfannemüller

Universität Mannheim

martin.pfannemueller@uni-mannheim.de

Christian Becker

Universität Mannheim

christian.becker@uni-mannheim.de

## ABSTRACT

Today's adaptive software systems (i) are often highly configurable product lines, exhibiting hundreds of potentially conflicting configuration options; (ii) are context dependent, forcing the system to reconfigure to ever-changing contextual situations at runtime; (iii) need to fulfill context-dependent performance goals by optimizing measurable nonfunctional properties. Usually, a large number of consistent configurations exists for a given context, and each consistent configuration may perform differently with regard to the current context and performance goal(s). Therefore, it is crucial to consider nonfunctional properties for identifying an appropriate configuration. Existing black-box approaches for estimating the performance of configurations provide no means for determining context-sensitive reconfiguration decisions at runtime that are both consistent and optimal, and hardly allow for combining multiple context-dependent quality goals. In this paper, we propose a comprehensive approach based on Dynamic Software Product Lines (DSPL) for obtaining consistent and optimal reconfiguration decisions. We use training data obtained from simulations to learn performance-influence models. A novel integrated runtime representation captures both consistency properties and the learned performance-influence models. Our solution provides the flexibility to define multiple context-dependent performance goals. We have implemented our approach as a standalone component. Based on an Internet-of-Things case study using adaptive wireless sensor networks, we evaluate our approach with regard to effectiveness, efficiency, and applicability.

## CCS CONCEPTS

• **Software and its engineering** → **Software product lines**;  
• **Mathematics of computing** → *Integer programming*; • **Networks** → Logical / virtual topologies; • **Computing methodologies** → Supervised learning by regression;

## KEYWORDS

Dynamic Software Product Lines, Performance-Influence Models, Machine Learning

## ACM Reference Format:

Markus Weckesser, Roland Kluge, Martin Pfannemüller, Michael Matthé, Andy Schürr, and Christian Becker. 2018. Optimal Reconfiguration of Dynamic Software Product Lines Based on Performance-Influence Models. In *SPLC '18: 22nd International Systems and Software Product Line Conference, September 10–14, 2018, Gothenburg, Sweden*. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3233027.3233030>

## 1 INTRODUCTION

Nowadays, software systems are getting increasingly configurable, allowing users and developers to customize the system to achieve certain functionality and improve the performance. Such software systems may exhibit hundreds of—often conflicting—configuration options resulting in a large configuration space [36]. The situation becomes even more complicated for self-adaptive systems (e.g., adaptive Wireless Sensor Networks (WSNs) [4]), which need to reconfigure continuously at runtime to meet consistency requirements imposed by the current context (e.g., enable a reliable transport protocol for a safety-critical application).

Besides these functional requirements, an adaptive system should minimize or maximize nonfunctional properties (e.g., latency, fairness), as stated by the current performance goal(s). Performance goals are often conflicting (e.g., minimizing the energy consumption in WSNs, but limiting the increase in latency) and context dependent, and, thus, may change at runtime. For example, when a safety-critical WSN switches to emergency mode, latency should be minimized, whereas, in normal operation mode, energy consumption should be minimized. In this regard, finding a configuration that is both consistent and performance-improving for changing

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

*SPLC '18, September 10–14, 2018, Gothenburg, Sweden*

© 2018 Association for Computing Machinery.

ACM ISBN 978-1-4503-6464-5/18/09...\$15.00

<https://doi.org/10.1145/3233027.3233030>

performance goals within a given context is a relevant and challenging task.

Dynamic Software Product Line (DSPL) techniques (e.g., context feature models [29]) are an established means to deal with the complexity of the large number of configuration options, capturing runtime variability and context dependencies of a configurable software system [15]. This makes DSPL a promising approach to specify, evaluate, and ensure consistency properties of adaptive systems at design and runtime. Usually, a large number of consistent configurations exists for a given context. Therefore, it is crucial to additionally consider performance goals to identify appropriate reconfiguration decisions. However, assessing the influence of a configuration on the system performance is a hard task due to interacting configuration options (e.g., combinations of routing and transport algorithms).

Recently, black-box performance models were proposed to estimate the performance influence in terms of measurable non-functional properties on the overall self-adaptive software system [14, 19, 37]. In particular, performance-influence models [36] can both estimate the influence of a specific configuration on performance in terms of nonfunctional properties as well as explicitly represent interactions between configuration options. However, in isolation, they (i) do not allow to determine an optimal configuration for a given context, (ii) do not provide means for determining both consistent and optimal reconfiguration decisions in regard to a given context, and (iii) provide no flexibility to specify context-dependent performance goals that may change at runtime. Existing approaches for black-box performance prediction are, therefore, inappropriate for planning of reconfigurations in self-adaptive software systems.

In this paper, we propose a *comprehensive DSPL-based approach for obtaining optimal reconfiguration decisions while maintaining a consistent system state*. Our solution enables design-time analysis and validation, resulting in a consistent reconfiguration model. We use training data obtained from simulations (and potentially real-world systems) to learn performance-influence models at design time as shown in [36]. In a novel runtime representation, we capture both, consistency properties, provided by context feature models, and learned performance-influence models for planning consistent and optimal reconfiguration decisions at runtime. Our solution provides flexibility to define context-dependent performance goals that are (ex-)changeable at runtime. Furthermore, we provide an implementation of our approach in a complete MAPE-cycle and show applicability, effectiveness, and efficiency of our solution by means of a real-world case study based on a self-adaptive system for topology control of WSNs. In summary, the major contributions of this paper are as follows:

- We introduce a novel DSPL-based runtime representation for capturing functional consistency properties and nonfunctional performance influences of configurations.
- We develop and implement an approach for learning the novel runtime representation from simulation training data and deriving optimal reconfiguration decisions at runtime based on the learned performance-influence models.

- We demonstrate applicability, effectiveness, and efficiency of our solution with an experimental evaluation on a real-world scenario.

## 2 BACKGROUND AND MOTIVATION

In this section, we motivate and outline the challenges that our approach tackles using an Internet of Things example.

### 2.1 Running Example: Adaptive WSNs

As a running example, we consider adaptive wireless sensor networks (WSN), which are an integral part of the emerging Internet of Things [13]. A WSN consists of a large number of sensor nodes that reliably detect particular events (e.g., rapidly increasing water levels in a flood warning system [4] or animals entering an observed area [9]). The *topology* of a WSN is a graph whose nodes represent the sensor devices and whose weighted edges describe communication links between sensor devices. Modifying the topology of a WSN influences its performance. For instance, in a dense network topology, information is spread with low latency and high robustness at the expense of high energy consumption and interference. *Topology control algorithms* [34] build on this observation and deactivate certain links to maintain a balance between reducing energy consumption and further performance goals (e.g., limiting latency).

Over the years, dozens of topology control algorithms have been proposed (see, e.g., [34]). Besides the topology control algorithm, sensor nodes have dozens of further configuration options (e.g., the routing and transport protocols). Figure 1 outlines the possible configuration options and context properties of a sensor node in our example. We consider the following representative topology control algorithms: Maxpower, LMST [25],  $l^*$ -kTC [42], e-kTC [22], Yao [26]. A topology control algorithm may have configuration options (e.g.,  $k$  of  $l^*$ -kTC). Sensor nodes may be mobile, which causes the topology to change continuously and requires the active topology control algorithm to run periodically as specified by the Interval option.

We consider three different generic application scenarios of WSNs: the WSN nodes report either all data to a dedicated base-station node (many-to-one communication, Datacollection) or exchange data with individual target nodes (e.g., to compensate for lost data points, PointToPoint and Gossip). The PointToPoint and Gossip applications represent a one-to-one communication pattern, where each node—deterministically (PointToPoint) or probabilistically (Gossip)—chooses a dedicated target node and transmits data to it. Additionally, sensor nodes may be mobile (e.g., when attached to an animal), captured by the property MobilitySpeed (in  $\frac{m}{s}$ ). The properties WorldSize (i.e., the side length of the observed area) and TopologyDensity (i.e., the mean node degree) are known to influence the performance of a topology control algorithm [12].

The following characteristics make reconfiguring WSNs especially challenging in practice. (i) **Large configuration space:** A large number of topology control algorithms exists, each potentially providing several configuration options. Configuration options of topology control algorithms interact, are mutual exclusive, or require specific communication mechanism to be active. To maintain

a good performance, topology control algorithms need to be reconfigured and even replaced at runtime. (ii) **Strong context dependencies:** The performance of a topology control algorithm depends considerably on the context (e.g., the current Scenario). Certain combinations of configuration options may be inapplicable in a particular context, leading to potentially inconsistent system states in the presence of dynamic environmental conditions (e.g., nodes joining/leaving the network). (iii) **Dynamic performance goals:** A change of the context may imply a change of the pursued quality goal. For example, the energy consumption of a flood detection system should be minimized in Normal operation mode, whereas latency should be minimized in Emergency mode. (iv) **Costly acquisition of reconfiguration knowledge:** Learning influences of contexts and system configurations on nonfunctional properties of a real system might be very expensive in practice due to the high number of configurations, or even infeasible for unlikely contexts.

## 2.2 Runtime Variability

*Dynamic Software Product Lines* (DSPL) constitute a methodology to capture a large configuration space concisely and to specify runtime variability of self-adaptive systems [4, 16]. Existing DSPL approaches provide means to specify self-adaptability in terms of valid configurations with reconfiguration decisions triggered at runtime. *Feature models* are used to specify the configuration space of the self-adaptive system during domain analysis. A *feature* refers to a domain abstraction from the problem space of the DSPL, corresponding to a binary configuration option of the self-adaptive system, which is either *selected* or *deselected*. A *configuration* is a set of selected features and corresponds to a valid system state.

*Feature diagrams* as part of the *Feature-Oriented Domain Analysis* (FODA) [20] restrict the configuration space by capturing dependencies between features. Figure 1 depicts the feature diagram specifying the configuration space of our running example. Features are organized in a *tree structure* to represent decomposition relationships between a parent feature and its children. For example, a System consists of the child-features Transport and Topology-ControlAlgorithm. A *decomposition relationship* between a parent feature and its children has one of the following types: *mandatory*, *optional*, *or*, and *alternative*. For example, the Transport layer can be configured with either UDP or TCP. A *cross-tree constraint* describes a relationship between features outside the tree structure and is either of type *require* or *exclude*. For example, if Emergency is selected, the Transport layer must be configured with UDP.

*Feature attributes* provide means to capture nonfunctional characteristics of features that can be measured [7]. Each attribute belongs to a *domain*, which represents the space of possible values of the attribute [2]. In this paper, we consider discrete (e.g., integer for Interval) and continuous domains (e.g., real for parameter k). A domain is further characterized by an interval that specifies lower and upper bounds (e.g.,  $\text{coneCount} \in \{8, \dots, 16\}$ ). Relationships between attribute values and features usually constrain the configuration space. We consider constraints of the form

$$e_{lhs} [\text{requires} \mid \text{excludes}] e_{rhs},$$

where  $e_{lhs}$ ,  $e_{rhs}$  are relational constraints of the form  $[f|a][< \mid > \mid =] v$  with the (binary) feature  $f$ , the attribute  $a$ , and the value  $v$ , which is within the domain of  $f$  or  $a$ , respectively. In our example,

Maxpower must be selected if the value of WorldSize is less than or equal to 300 m.

Context variability plays a key role for DSPLs, as system configurations must adapt to changes in the context [6]. *Context feature models* (CFMs) allow to model relevant properties of the context as *context features* and to specify their dependencies to *system features* using cross-tree constraints [17]. In Figure 1, we organize system features and context features in separate branches of the feature model. Context features reside below the Context feature and system features reside below the System feature. Features of the context branch are selected to reflect a given context. The *context configuration* is the partial configuration that represents all selected features of the context branch. Accordingly, the *system configuration* is the partial configuration that represents the system branch.

When the context changes, the context configuration is adjusted accordingly, and the system configuration may need to be adapted. For example, if transport is configured with TCP and the context configuration changes from Normal to Emergency, the requirement constraint between the context feature Emergency and the system feature UDP forces a selection of UDP and a deselection of TCP.

## 2.3 Challenges

The observations regarding the running example lead us to the following challenges that we tackle in this paper.

### (1) Joint consideration of performance goals and consistency.

In our example, for a context configuration consisting of Emergency,  $\text{WorldSize} = 1000$ ,  $\text{TopologyDensity} = 100$ , and Gossip, infinitely many consistent system configurations meet the context requirements due to real-valued attributes. As the performance of a system configuration in our characterized scenario depends strongly on its context, we have to consider nonfunctional properties for choosing an appropriate configuration. The first challenge is to find, for a given context configuration, a consistent but also a performance-improving/-optimizing system configuration. To this end, our approach provides a runtime model to incorporate both consistency and nonfunctional influences between context and system features.

### (2) Context-dependent performance goals.

In our example, if the Mode changes from Normal to Emergency mode, the performance goal changes from minimizing the mean energy consumption of the network to minimizing the latency of transferred messages. Here, changing the performance goal should lead to an adaptation of the system configuration to reduce path lengths in the topology. Often, we would like to pursue multiple quality goals at the same time with adjustable priorities (e.g., minimizing energy consumption while limiting the increase in latency). The second challenge is to support multiple (possibly contradicting) performance goals during reconfigurations. To this end, our approach allows to specify potentially conflicting quality goals that are exchangeable at runtime.

### (3) Uniform representation of reconfiguration knowledge.

We need to acquire and specify knowledge about influences between context and system features. Given the complexity and size of the configuration space, we have to build up this knowledge



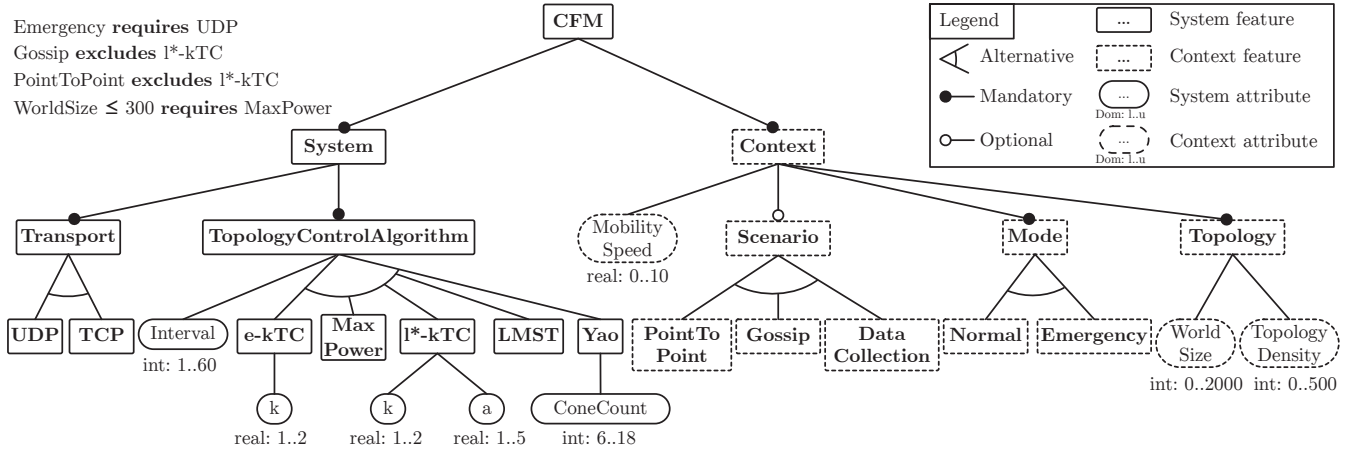


Figure 1: Context Feature Model (CFM) of the running example

primarily using machine learning. Therefore, acquiring the necessary training data from a real-world system is often expensive or infeasible. Simulators mimic the behavior of real-world systems and make generating large training datasets affordable. Incorporating domain knowledge in addition to the (generated and collected) training dataset should be possible. The third challenge is to facilitate combining reconfiguration knowledge obtained from simulations, real-world system, and domain experts with the variability knowledge, encoded using feature models. To this end, our approach builds on an integrated runtime model that represents knowledge about valid reconfigurations, context dependencies, and influences between context and system features.

### 3 OPTIMAL RECONFIGURATION BASED ON PERFORMANCE-INFLUENCE MODELS

In this section we present our approach to address the challenges stated in Section 2.3.

#### 3.1 Overview of Approach

Figure 2 provides an overview of our approach, which is based on the MAPE-K framework [21]. MAPE-K separates the adaptation logic of a system into four phases—*Monitoring*, *Analysis*, *Planning*, *Execution*—that are executed in the given order and access a shared *Knowledge* component. In the following, we outline the role of each MAPE-K phase in our approach.

**Sensor, Monitoring & Analysis.** The sensor collects uninterpreted contextual data from the managed resource. During the monitoring phase (see ①), the sensor data is validated and, during the analysis phase, transformed into a context configuration. The context configuration is then passed to the planning component. Additionally, the monitoring component collects samples of measured nonfunctional properties at runtime and stores them in a repository for learning influences of configurations on performance at design time (see ②).

**Planning.** During the planning phase, the planning component determines a consistent system configuration based on the context

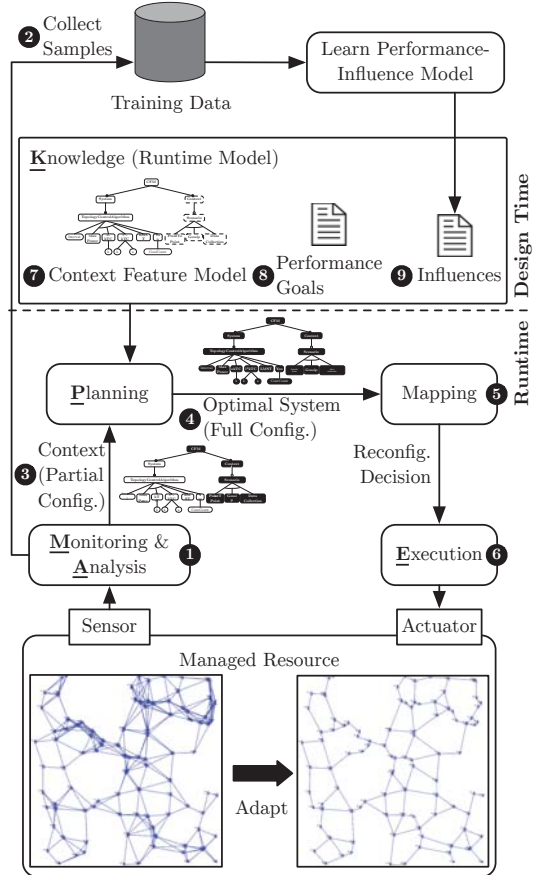


Figure 2: Overview of the architecture

configuration (see ③). Specifically, we take into account consistency properties from the context feature model, learned influences of configurations on nonfunctional properties, and performance goals. To this end, the context feature model is transformed as well

as the performance-influence model into a mathematical optimization problem. Depending on the context, the optimization objective is adjusted to represent the context-dependent performance goals. Additionally, we incorporate constraints that fix the problem variables corresponding to the current context configuration. A solution of the resulting optimization problem corresponds to an optimal complete configuration (see 4). We assume that, for each context configuration, at least one consistent system configuration exists. To guarantee this property at runtime, we perform design-time consistency checks as shown in [3, 44].

**Execution & Actuator.** Prior to the execution phase, the resulting configuration is mapped to the target representation of the managed resource constituting the reconfiguration decision (see 5). In our example, the target representation corresponds to components implementing the topology control algorithms and parameters defining their behavior. During the execution phase (see 6), the reconfiguration decision is passed to the actuator, modifying the state of the managed resource (e.g., by exchanging the active topology control algorithm).

**Knowledge.** Throughout the MAPE phases, the knowledge component represents the reconfiguration knowledge of the adaptation logic and comprises (i) the context feature model (see 7), specifying consistency properties of the system, (ii) the (context-dependent) performance goals in terms of nonfunctional properties, which need to be maximized or minimized (see 8), (iii) the performance-influence models (see 9), learned at design-time from simulation-based, real-world, and manually specified data. A separate performance-influence model is learned for each nonfunctional property. The context feature model and the performance goals are specified at design time.

### 3.2 Learning Performance-Influence Models

After the preceding high-level overview of our approach, we now explain in detail how we learn and represent dependencies of configurations on nonfunctional properties in form of performance-influence models. Learned performance-influence models allow to assess performance for a particular configuration in terms of a single nonfunctional property. However, they do not allow to derive an optimal configuration for a given context. To this end, our approach encodes information from multiple learned performance-influence models as well as consistency information as a mathematical optimization problem. We begin with describing the learning process for individual performance goals and later explain how multiple performance goals can be combined. To learn dependencies between features and their mutual influence on a measured nonfunctional property, we need to consider a configuration space with  $n$  dimensions  $X = \text{Dom}(x_1) \times \dots \times \text{Dom}(x_n)$ , where the vector  $\mathbf{x} \in X$  represents a particular configuration consisting of  $n$  entries  $x_i$ . An entry  $x_i$  either refers to a feature (i.e.,  $\text{Dom}(x_i) = \{0, 1\}$ ), to a bounded real valued attribute (i.e.,  $\text{Dom}(x_i) = \{x \in \mathbb{R} \mid l_i \leq x_i \leq u_i\}$ ), or to a bounded integer valued attribute (i.e.,  $\text{Dom}(x_i) = \{x_i \in \mathbb{N} \mid l_i \leq x_i \leq u_i\}$ ).

For each considered nonfunctional property, we obtain a *performance-influence model*  $\Pi$  by applying an incremental learning algorithm presented in [36]. We learn separate performance-influence models for each nonfunctional property (e.g., energy consumption, latency). In our example, a performance goal is to minimize the

energy consumption, which causes a topology control algorithm to reduce the number of communication links whereas a conflicting performance goal is to minimize the latency in the overall topology, which causes a topology control algorithm to increase the number of communication links. Hence, depending on the considered nonfunctional property, we want to either minimize or maximize a performance goal.

The incremental learning algorithm is based on a stepwise linear regression, estimating the performance  $\Pi(\mathbf{x})$  of a particular configuration as a sum of performance-influence terms. The input of our learning algorithm is a set of sampled configuration vectors  $\mathbf{x}_1, \dots, \mathbf{x}_k$  and corresponding measured values of the nonfunctional property  $\hat{\Pi}(\mathbf{x}_1), \dots, \hat{\Pi}(\mathbf{x}_k)$  (e.g., the measured latency for each sampled configuration). The samples originate from a simulator or real-world system. An estimated performance-influence model for a specific nonfunctional property is represented by equations of the form

$$\Pi(\mathbf{x}) = \sum_{I=\{p, \dots, q\} \in \mathcal{I}} \Phi_I(x_p, \dots, x_q), \quad \mathcal{I} = 2^{\{1, \dots, n\}}.$$

The *index powerset*  $\mathcal{I}$  contains all subsets of indices  $i$  where  $1 \leq i \leq n$  of a configuration  $\mathbf{x} \in X$ . The *performance-influence term*  $\Phi_I$  captures mutual influences of multiple configuration options  $x_p, \dots, x_q$ . The term  $\Phi_\emptyset$  represents a constant base performance value that is independent of any configuration option. Note that the size for  $\mathcal{I}$  grows exponentially with the size of the feature model. For this reason and to avoid overfitting while learning the performance-influence model, it is sensible to limit the cardinality of the index subsets contained in  $\mathcal{I}$ . More precisely, to reflect (at most)  $t$ -wise feature interactions, we may use the following index powerset:  $\mathcal{I}_t = \{I \in \mathcal{I} : |I| \leq t\}$ .

A performance-influence term has the form

$$\Phi_I(x_p, \dots, x_q) = \omega_I \cdot x_p^{\tau_p} \cdot \dots \cdot x_q^{\tau_q}.$$

The real-valued coefficient  $\omega_I$  weights the product of interacting configuration options  $x_p, \dots, x_q$ . The exponents  $\tau_p, \dots, \tau_q$  control the weight of their corresponding configuration options in the term. In our example, a possible performance-influence term w.r.t. latency could be

$$\Phi_{\text{ex}} = 2 \cdot e\text{-kTC} \cdot \text{TopologyDensity}^2 \cdot k^2,$$

which denotes that the latency tends to increase with increasing TopologyDensity and growing parameter  $k$  iff  $e\text{-kTC}$  is selected.

The performance-influence terms encode the following two types of information. (i) *Nonlinear influences among configuration options:* a performance-influence term represents interactions of configuration options explicitly. Whenever all configuration options occurring in performance-influence term are selected, the system performance is expected to be influenced in a nonlinear manner (depending on the number of configuration options). We are especially interested in interactions between system and context features. When the context changes and a newly selected context feature is part of a previously learned performance-influence term, we should consider planning a reconfiguration as we know that the system performance is influenced by the context change. (ii) *Quantifiable influences among configuration options:* we can assess the expected influence on the system performance from the weights

$\omega_I$ , the domains of the interacting attributes, and their exponents  $\tau_p, \dots, \tau_q$ . Terms with large weights have a potentially significant influence on the system performance. Accordingly, attributes with a large range between their lower and upper bound of their domain and with larger exponents potentially have a more significant influence on the system performance.

The performance-influence models are learned at design-time for each nonfunctional property separately and allow to assess the (expected) performance of a particular configuration. However, the learned representation of performance influences does not allow to compute an optimal configuration for a given context. Still, a reconfiguration needs to maintain consistency, as specified by the context feature model. To this end, in the next step, we show how to combine consistency properties and the learned performance-influence models in a runtime model to perform optimal reconfigurations while maintaining consistency.

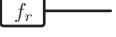


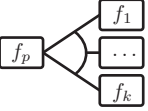
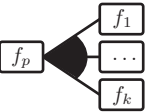
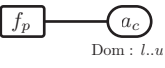
### 3.3 Runtime Model

Constraint solvers allow to derive consistent configurations from feature models with context properties [3]. This requires to formulate the reconfiguration problem as a constraint satisfaction problem [28]. However, to obtain optimal reconfigurations, we also have to encode the performance-influence models and the weighing of multiple performance goals. Therefore, the resulting formalization is the following optimization problem: For a given context, find a consistent configuration that is optimal with regard to the performance-influence model.

We formulate the problem as a mathematical optimization problem [45] to determine the optimal solutions using off-the-shelf mathematical programming solvers or to obtain near-optimal solutions using heuristics. A *mathematical optimization problem* consists of a set of linear inequalities on a set of  $k$  integer- and real-valued *decision variables*. The resulting convex hull forms the *feasible region* within a  $k$ -dimensional search space. The *objective function* states that either lower (minimum) or upper (maximum) boundary value for decision variables should be found by a solver.

**Encoding Consistency Properties.** We now describe how each feature and attribute of the context feature model is encoded into the optimization problem. For each feature, we introduce a *feature (decision) variable*  $f_i \in \{0, 1\}$ , denoting whether the feature corresponding to  $f_i$  is selected in a configuration. For each attribute, we introduce an *attribute (decision) variable*  $a_i \in \mathbb{R}$  for attributes with real-valued domains and  $a_i \in \mathbb{N}$  for attributes with integer domains, denoting the selected value of an attribute. The resulting formulation is a *Mixed Integer Linear Programming* (MILP) problem [45] because we use both integer and real-valued variables.

To encode consistency properties of the context feature model, we use a formulation similar to the SAT-based encoding in [8] that we adjusted to conform to our MILP problem. Table 1 lists the encoding scheme for each type of consistency property. For binary variables used in the propositional terms, we assume that a feature is selected iff  $f_i = 1$ . The Boolean expressions for encoding propositional terms are intermediate representations because constraints in MILP need to be linear inequalities. Therefore, we provide a MILP encoding for propositional terms comprising Boolean linear

Property	FM Syntax	Constraint
Root		$f_r = \text{true}$
Optional		$f_c \Rightarrow f_p$
Mandatory		$(f_c \Rightarrow f_p) \wedge (f_p \Rightarrow f_c)$
Alt.-Group		$f_p \Rightarrow (f_1 + \dots + f_k = 1)$
Or-Group		$f_p \Rightarrow (f_1 + \dots + f_k \geq 1)$
Attribute		$f_p \Rightarrow (l \leq a_c \wedge a_c \leq u)$
Require	$e_{\text{lhs}}$ <b>requires</b> $e_{\text{rhs}}$	$e_{\text{lhs}} \Rightarrow e_{\text{rhs}}$
Exclude	$e_{\text{lhs}}$ <b>excludes</b> $e_{\text{rhs}}$	$\neg(e_{\text{lhs}} \wedge e_{\text{rhs}})$

**Table 1: Encoding consistency properties into constraints.**

arithmetic expressions. We first transform the depicted propositional terms to their corresponding conjunctive normal form (CNF) and linearize products of binary variables by using special ordered sets [1].

All constraints described so far remain unchanged at runtime. Additionally, we add further context-specific constraints, which may differ in each execution of the MAPE loop. The context-specific constraints are derived from the context configuration, which is calculated during the analysis phase. This context configuration results in adding additional constraints of the form  $f_i = 1$  (i.e., if context feature  $f_i$  is selected), and  $a_i = v$ , (i.e., the value  $v$  is assigned to attribute variable  $a_i$ ).

**Encoding Performance-Influence Terms.** In this step, we illustrate how the learned performance-influence model is integrated into the MILP problem to reflect the current performance goal(s). The basic idea of our approach is to transform each performance-influence term  $\Phi_I$  into a partial linear objective function, reflecting the influence of the performance-influence term on the nonfunctional property value. The objective function of the entire performance-influence model is the sum over all partial objective functions. The nonfunctional property determines whether the objective function should be minimized (e.g., for energy consumption) or maximized (e.g., for fairness of resource consumption).

In the first transformation step, we introduce for each performance-influence term a *presence condition*, which specifies whether the performance-influence term may contribute to the overall nonfunctional property value. According to Table 1, an attribute  $a_c$  is selected and lies within its bounds iff its parent feature  $f_p$  is selected. Thus, a term  $\Phi_I$  may only influence the system performance

if the presence condition  $f_i \wedge \dots \wedge f_j$  evaluates to *true*, where each  $f_i, \dots, f_j$  is either a feature or, in case of an attribute, the parent feature of an attribute. The so refined performance-influence term  $\Phi'_I$  has the form

$$\Phi'_I = \begin{cases} \omega_I \cdot a_r^{\tau_r} \cdot \dots \cdot a_s^{\tau_s} & \text{if } f_i \wedge \dots \wedge f_j = \text{true} \\ 0 & \text{otherwise,} \end{cases}$$

where the configuration options  $a_r, \dots, a_s$  are the attribute values that remain when removing all  $f_i, \dots, f_j$  from the arguments of the original performance-influence term  $\Phi_I$ . In our example, the presence condition is  $\text{e-kTC} \wedge \text{Topology} = \text{e-kTC}$  because Topology is mandatory. The refined sample performance-influence term is now

$$\Phi'_{\text{ex}} = \begin{cases} 2 \cdot \text{TopologyDensity}^2 \cdot k^2 & \text{if e-kTC} = \text{true} \\ 0 & \text{otherwise.} \end{cases}$$

Still, the refined performance-influence terms are nonlinear due to the multiplication of decision variables and the exponents  $\tau_r, \dots, \tau_s$ . Therefore, we conduct a *log-transformation* [10] as second transformation step. More precisely, we introduce auxiliary variables  $a' \in \mathbb{R}$  for each integer and real-valued attribute  $a$  with  $a' = \log(a + \delta)$ , where the translation constant  $\delta$  is calculated as follows:

$$\delta = \begin{cases} 1 - \min_{I \in \mathcal{I}} \omega_I & \text{if } \min_{I \in \mathcal{I}} \omega_I < 0 \\ 1 & \text{otherwise.} \end{cases}$$

This definition of  $\delta$  ensures that all performance-influence terms are transformed uniformly and that the transformed terms always evaluate to values greater than or equal to 1. The constraints that encode the domain bounds of attribute  $a$  are transformed as well; for an attribute  $a$  with  $l \leq a \leq u$  and parent feature  $f_p$ , the resulting constraint is:

$$f_p \Rightarrow (\log(l + \delta) \leq a' \wedge a' \leq \log(u + \delta)).$$

Finally, by applying the log-transformation to the entire performance-influence term  $\Phi'_I$ , we retrieve a linear term  $\Phi''_I$  of the form

$$\Phi''_I = \begin{cases} \log(\Phi'_I(a_r, \dots, a_s)) = \\ \log(\omega_I + \delta) + \tau_r a'_r + \dots + \tau_s a'_s & \text{if } f_i \wedge \dots \wedge f_j = \text{true} \\ 0 & \text{otherwise.} \end{cases}$$

Again,  $\Phi''_I$  is greater than 0 only if the presence condition is *true*. The resulting log-transformed term for our example is:

$$\Phi''_{\text{ex}} = \begin{cases} \log(2) + 2 \cdot \text{TopologyDensity}' + 2 \cdot k' & \text{if e-kTC} = \text{true} \\ 0 & \text{otherwise.} \end{cases}$$

The log-transformed performance-influence terms  $\Phi''_I$  directly correspond to summands in the objective function  $\Pi''$  of the resulting MILP problem, and the presence conditions are transformed into appropriate constraints of the MILP problem.

**Encoding Multiple Performance Goals.** So far, we only considered one performance goal. However, our solution can be easily extended to combine and weight multiple performance goals. To encode  $m \geq 1$  performance goals, we first need to learn  $m$  performance-influence models and, then, translate each performance-model into the mathematical optimization problem with objective function equivalent to  $\Pi''_m$ , as shown before. The *combined*

*objective function*  $\Pi''$  can then be represented as linear combinations of the individual objective functions:

$$\Pi''(\mathbf{x}) = \gamma_1 \cdot \Pi''_1(\mathbf{x}) + \dots + \gamma_m \cdot \Pi''_m(\mathbf{x}).$$

The *objective-weighting coefficients*  $\gamma_k, 1 \leq k \leq m$  correspond to scalarization factors for weighting the different performance goals [27]. An objective function for our example could be

$$\Pi''(\mathbf{x}) = \gamma_1 \cdot \Pi''_{\text{latency}}(\mathbf{x}) + \gamma_2 \cdot \Pi''_{\text{energy}}(\mathbf{x})$$

where the conflicting performance goals regarding the nonfunctional properties latency and energy consumption are combined. Our approach supports adjusting coefficients  $\gamma_k$  at runtime to represent context-dependent performance goals by adding objective-weighting constraints between context features and  $\gamma_k$ . For example, for a normal system state (i.e., whenever mode Normal is selected) the energy consumption needs to be minimized, whereas for an emergency system state (i.e., whenever mode Emergency is selected) the latency needs to be minimized. By adding the constraints  $\text{Emergency} \Rightarrow (\gamma_1 = 1)$  and  $\text{Normal} \Rightarrow (\gamma_2 = 1)$  with  $\gamma_1 + \gamma_2 = 1$  this and similar scenarios, as shown in [4], can be specified.

## 4 EVALUATION

In this section, we present the results of evaluating our approach w.r.t. effectiveness, efficiency, and applicability using adaptive WSNs as an evaluation scenario. We provide supplementary material of this evaluation online<sup>1</sup>. We answer the following research questions: **RQ1-Effectiveness:** How much does our approach improve the system performance w.r.t. the desired performance goal in terms of nonfunctional properties? What is the influence of the performance-influence model size on the system performance?

**RQ2-Efficiency:** How does the size of the performance-influence model affect the runtime of the planning component?

**RQ3-Applicability:** Is the approach applicable to self-adaptive systems in general?

### 4.1 Experimental Setup

The technical setup of our evaluation is an extension of the rapid-evaluation environment described in [22]. The simulation environment consists of the Java-based wireless network simulator SIMONSTRATOR 2.5 [32] and the model-driven engineering tool eMOFLON 2.32.0 [24], used for implementing the topology control algorithms. For this paper, we extended the simulation environment with an adaptation logic, which is loosely coupled via JSON-based interfaces for sensing and effecting and built using FESAS 1.0.1<sup>2</sup>, a framework for developing self-adaptive systems [23]. All measurements were run on a 64-bit Windows 7 workstation, equipped with an Intel i7-2600 CPU (2 × 3.7 GHz) and 8 GB of RAM, with 2 GB being assigned to the simulator.

We evaluate our system w.r.t. the nonfunctional property *latency*. Reducing latency is a key performance goal for (wireless) communication systems because low latency ensures that all nodes have an up-to-date view of the network.

<sup>1</sup>Supplementary material: <https://github.com/Echtzeitsysteme/splc2018>

<sup>2</sup>Project site: <https://fesas.bwl.uni-mannheim.de/>

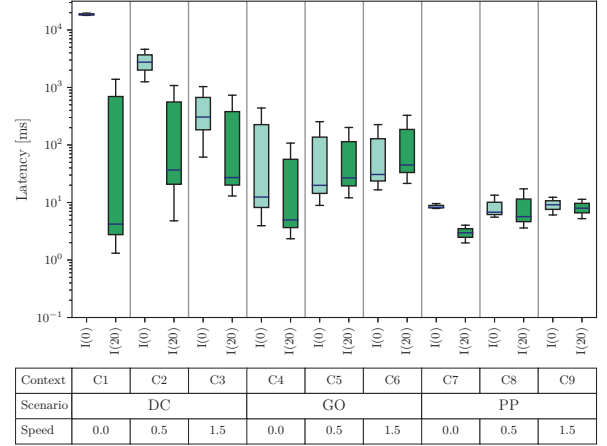


## 4.2 Acquiring Training and Test Datasets

The data of this evaluation resulted from two batches of simulation runs. The first batch run produced the raw training data for the incremental learner. The second batch run served to compare our approach against the baseline system w.r.t. effectiveness and efficiency. Given a context configuration, the baseline system produces a random valid configuration.

The *training dataset* resulted from 6 125 simulation runs, each representing a fixed configuration (i.e., system and context were preconfigured). For each simulation run, we determined the mean latency. All training simulation runs took ca. 412 h of CPU time. We used SPL CONQUEROR [37] to obtain a corresponding performance-influence model and configured the learning to stop if either (i) the improvement per round dropped below  $10^{-8}$ , (ii) the training took more than 100 rounds, or (iii) the duration exceeded 60 min. Accordingly, the training finished due to the time limit after 64 min and 38 iterations. As maximum size of a performance-influence term, we used the default value  $t = 4$ . In the following,  $N$  denotes the number of training iterations that produced the performance-influence model that is used by the planner of the system under evaluation. Starting with a performance-influence model consisting of two terms, the learning algorithm adds one performance-influence term to the performance-influence model in each iteration. Therefore, after  $N$  iterations, the performance-influence model consists of  $N+1$  terms. For  $N > 0$ ,  $I(N)$  denotes the MILP-based planner with a performance-influence model of size  $N + 1$ , and  $I(0)$  denotes the baseline system, which has an empty performance-influence model. For RQ2, we also measured the runtime of a SAT-based planner  $M(0)$ , using MINISAT [40]. Using SAT, we can only encode consistency constraints of the CFM that relate to Boolean features, but we can encode neither complex attribute constraints of the CFM nor constraints learned from the performance-influence model efficiently. In total, we considered the following two baseline and seven planners:  $M(0)$ ,  $I(0)$ ,  $I(5)$ ,  $I(10)$ ,  $I(15)$ ,  $I(20)$ ,  $I(25)$ ,  $I(30)$ ,  $I(35)$ .

The *test dataset* resulted from 1 215 simulation runs, corresponding to five simulation runs per combination of context (Scenario, MobilitySpeed, WorldSize, TopologyDensity) and planner. The context configurations resulted from combining the three possible values of Scenario with  $\text{WorldSize} \in \{300 \text{ m}, 900 \text{ m}\}$  and the average movement speed  $\text{MobilitySpeed} \in \{0 \frac{\text{m}}{\text{s}}, 0.5 \frac{\text{m}}{\text{s}}, 1.5 \frac{\text{m}}{\text{s}}\}$  for the Gauss-Markov model [5]. The latency value of a simulation run is the mean latency of all packets (in ms). Measurements started after a warm-up time of 5 min simulation time. In each simulation run, 100 fully-charged nodes were distributed randomly onto a square region. The value of TopologyDensity was measured after placing the nodes. A central *planner node* conducted the planning, and all nodes reconfigured to the planner decision. For Datacollection, the base station was the planner node. For PointToPoint and Gossip, the planner node was chosen randomly at the beginning of the simulation. Reconfiguration decisions were performed immediately prior to the invocation of the topology control algorithm. As MILP solver, we used CPLEX 12.7.1 (Academic Edition) [18]. The network stack consisted of a IEEE 802.11b link layer, a global-knowledge-based routing layer, and a UDP transport layer. The transmission range was set to 130 m, and the state-based energy model mimicked a Broadcom BCM4329 chip and a 1300 mAh battery.



**Figure 3: Latency vs. context class for  $I(0)$  and  $I(20)$  (DC: Datacollection, GO: Gossip, PP: PointToPoint)**

## 4.3 RQ1: Effectiveness

With RQ1, we assess whether our approach achieves the chosen performance goal better than the baseline system  $I(0)$ . We first gather an overview of the performance of our system and then analyze the trade-off between performance and training cost.

**Latency vs. context class.** Figure 3 summarizes the latency for nine context classes  $C1, \dots, C9$  and provides an overview of the system performance. A context class aggregates the latency values of all evaluation runs sharing the same Scenario and MobilitySpeed. The x axis shows all nine combinations of Scenario and MobilitySpeed. The y axis has a logarithmic scale because, otherwise, the latency of our system could not be distinguished from 0 ms. For each context class, two boxplots summarize the latency values of the baseline system  $I(0)$  (left, lighter boxplot) and the MILP-based planner with a performance-influence model that results from 20 training iterations  $I(20)$  (right, darker boxplot). The boxplots follow the convention that (i) the blue horizontal line indicates the median value, (ii) the lower and upper caps mark the 25%- and 75%-percentile, (iii) the whiskers enclose all values within the 1.5 inter-quartile range, and (iv) outliers are shown as circles.

In seven of nine context classes, using  $I(20)$  exhibits a lower median latency compared to  $I(0)$ , with a reduction of latency between 11.7 % ( $C9$ ,  $I(0)$ : 9.03 ms,  $I(20)$ : 7.97 ms) and a factor of  $4.4 \cdot 10^3$  ( $C1$ ,  $I(0)$ :  $1.86 \times 10^4$  ms,  $I(20)$ : 4.21 ms). For two context classes (both having a Gossip scenario), the median latency of  $I(20)$  is larger: 32.4 % for  $C5$  ( $I(0)$ : 20.2 ms,  $I(20)$ : 26.8 ms) and 48.3 % for  $C6$  ( $I(0)$ : 30.3 ms,  $I(20)$ : 45.0 ms). The necessary logarithmic scale leads to a distorted impression of the ranges. For example, in  $C1$ , the inter-quartile range is  $1.79 \times 10^4$  ms to  $1.96 \times 10^4$  ms for  $I(0)$ , compared to 1.31 ms to  $1.39 \times 10^3$  ms for  $I(20)$ .

Judging by median values, our system improves upon the baseline system in most contexts. Still, especially for Datacollection and Gossip, the baseline and system boxes overlap for three of the seven context classes where  $I(20)$  is superior. In the remaining four context classes,  $I(20)$  is clearly superior. The reason for these



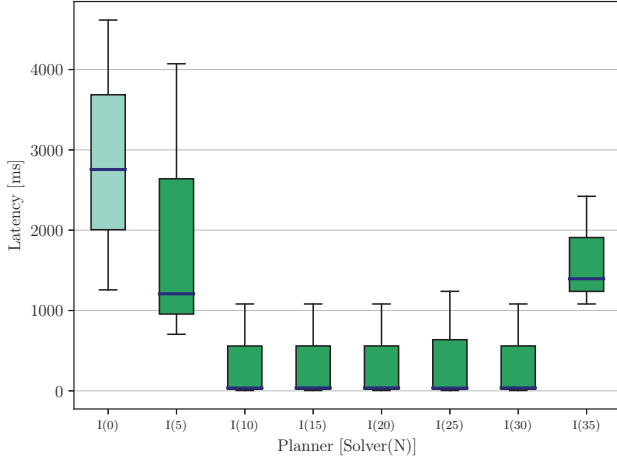


Figure 4: Latency vs. training cost for C2

observations is that each context class aggregates 120 simulation runs. An in-depth inspection of the results per configuration reveals a considerable lower spread (e.g., Figure 4). The differing median values for the different scenarios result from the underlying communication patterns (deterministic vs. probabilistic, many-to-one vs. one-to-one).

**Latency vs. training cost.** We now analyze the influence of varying the performance-influence model size on the system performance. To this end, we focus on the representative context class C2 (i.e., Datacollection and MobilitySpeed =  $0.5 \frac{m}{s}$ ). Figure 4 shows the latency for different values of  $N$  in C2. The third and forth boxplots of Figure 3 reappear as I(0) and I(20) in Figure 4.

The median latency decreases with increasing  $N$ , starting from  $2.77 \times 10^3$  ms for I(0), stabilizing at 35.5 ms for I(10) and 36.7 ms  $N \in \{15, \dots, 30\}$ , and then increasing again to  $1.4 \times 10^3$  ms for I(35). An inspection of the simulation log files reveals that the planner returns the exact same reconfiguration decisions for  $N \in \{15, \dots, 30\}$ , leading to stable overall simulation results.

These results allow for two notable conclusions. First, the incremental learning algorithm shows overfitting in C2 for large values of  $N$ , which is reasonable given that the corresponding performance-influence model has 35 terms. Second, the learning algorithm is surprisingly robust against varying the number of learning iterations, which is equivalent to  $N$ . Robustness can be observed for all remaining context classes exhibit robustness and overfitting for C3.

In summary, we answer RQ1 as follows: The results indicate that our system is able to improve upon a random baseline system in the majority of context classes, but the large spread per context class limits the significance of the results. Furthermore, the incremental learning algorithm shows overfitting, which is not surprising, but the optimal performance value is robust w.r.t. varying the  $N$ .

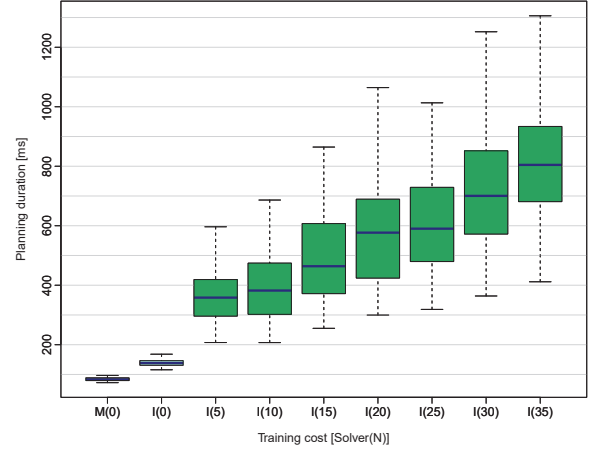


Figure 5: Planning duration vs. training cost

#### 4.4 RQ2: Efficiency

With RQ2, we assess the scalability of the planning component w.r.t. varying the size of the used performance-influence model. Assessing the runtime of the planning component is relevant because (i) reconfigurations happen periodically whereas the reconfiguration knowledge is acquired once during design time, (ii) the planner is independent of the underlying managed resource (e.g., the simulated sensor nodes), and (iii) our results show that the planning component requires the largest fraction of time of the MAPE cycle (ca. 82 %). Note that the number of constraints of the resulting ILP grows linearly with the number of performance-influence terms.

Figure 5 summarizes the mean planning duration for different planner setups. The x axis shows all nine planner setups that we introduced in Section 4.2. Each boxplot summarized the mean planning durations of all 135 simulation runs that used the given planner.

The results shown in Figure 5 indicate a clear dependency between the different planner setups and the planning duration. The median planning duration for the SAT-based baseline setup M(0) is 83.6 ms which is ca. 40 % less than the MILP baseline setup I(0), having a median of 138.5 ms. The median planning duration increases moderately from 358.3 ms for I(5) to 804.6 ms for I(35). The increased planning duration between setups M(0) and I(N) was to be expected because the MILP encoding considers (real-valued) attributes and objective functions, and is, therefore, richer in comparison to the SAT encoding. The increase in the planning duration between setups I(0) to I(35) was to be expected because, with increasing number of training iterations, the number of constraints in the underlying MILP problem also increases.

In summary, we answer RQ2 as follows: The results of the runtime measurements indicate that our system scales well with increasing performance-influence model sizes. The increase in runtime compared from a SAT-based planner to our MILP-based planner is in the same order of magnitude, which is a surprisingly positive finding given that the MILP encoding additionally captures complex attribute constraints and the performance goals of the performance-influence model.

#### 4.5 RQ3: Applicability and Threats to Validity

We now discuss the general applicability and potential threats to validity.

**General applicability.** The adaptation logic can be adjusted to new use cases by solely replacing the models in the knowledge component. To obtain training data for the learning algorithm, our approach is not limited to simulations. The input of the learning algorithm consists of traces, each mapping a configuration to a nonfunctional property value (e.g., real-world systems and domain knowledge). Besides, our approach allows to exchange and reconfigure the size of the used performance-influence model at runtime. In this paper, we extensively evaluated the case study of adaptive WSNs. Still, assessing a single case study is a threat to external validity w.r.t. applicability of our approach to other domains. We mitigated this threat (1) conceptually by using established formalisms (e.g., feature modeling, UML modeling, MILP) and (2) technically by implementing an external instead of an internal adaptation logic and by using generic exchange formats at the interfaces between the MAPE-K components (e.g., JSON, CSV, XML).

**Case study.** The following discussion of threats to validity focuses on the case study of this evaluation. As discussed in Section 4.4, the planner component requires the largest time fraction of the MAPE cycle because MILP solving is a computational-intensive task. Real WSN nodes are typically resource constrained, which complicates running an MILP-based planner on each device. This threat to external validity can be circumvented by using centralized or regional decision making and reconfiguration dissemination, e.g., regional clustering approaches for WSNs [43]. Evaluating our approach in a single simulation environment also imposes a threat to external validity. However, SIMONSTRATOR is a state-of-the-art network simulator that uses wireless models of the established ns-3 simulator [33] and has been used for diverse papers and dissertations<sup>3</sup>. This underpins our confidence in the salience of the simulation results regarding latency. To mitigate the construct threat of the inherent randomness in network simulations, we repeated all test dataset runs five times with different random seeds. Finally, only assessing our approach with one feature model could impose a threat to external validity. The feature model depicted in Figure 1 contains 20 features, 8 attributes, and 4 constraints, which is comparable to state-of-the-art case studies (e.g., [38, Table 1]).

## 5 RELATED WORK

In this section, we survey the related reconfiguration approaches. Previously, we used context feature models to obtain consistent configurations for a given context by employing SAT solvers [31]. In this work, we additionally consider learning of performance-influence models as well as the specification of performance goals and complex attribute constraints. Elkhodary et al. propose the FUSION framework to build self-adaptive software systems [11]. Similar to our approach, they learn the influence of adaptation decisions on performance goals as functions of features that allow for representing feature interactions. In contrast to our approach, the authors allow for fine-tuning parameters at runtime, but only consider binary features, do not support specifying contexts and context dependencies of features and attributes and provide no means to

specify context-dependent performance goals. In summary, their online-learning algorithm is complementary to our approach.

Sharifloo et al. propose a DSPL-based approach for tackling design-time uncertainty of context dynamics [35]. In contrast to our approach, they consider runtime evolution of the specifications while we assume that a specification is fixed at design-time. Sousa et al. propose an approach for extending DSPLs with temporal constraints to model the reconfiguration life cycle of a DSPL [41]. They allow to specify constraints on the reconfiguration behavior whereas, in our approach, reconfiguration from any configuration to any other configuration is possible. Both approaches only consider consistency between context and system configuration, and fail to support the selection of configurations by means of learned influences between configurations and measured nonfunctional properties.

In [30], Parra et al. propose to combine variability models with constraint-satisfaction-problem techniques to optimize context-aware adaptations. In contrast to our approach, they support neither complex dependencies between system and context configurations nor attributes as configuration options, and provide only limited support for deriving optimal configurations in large systems due to the combinatorial explosion in the number of configurations. In [39], Sinha et al. propose to reconfigure evolving systems automatically for deployment onto an available set of resources. Consistent system configurations are generated using SMT-based constraint resolution. Similar to our approach, attributes and complex dependencies between attributes are supported for deriving consistent configurations. In contrast to our work, performance goals are not considered.

## 6 CONCLUSION

We presented a comprehensive DSPL-based approach for obtaining optimal reconfiguration decisions while maintaining a consistent system state. Our approach uses training data obtained from simulators to learn performance-influence models at design time, and incorporates both consistency and nonfunctional influences between context and system features in a novel runtime representation. The runtime representation provides flexibility to define multiple context-dependent performance goals with regard to nonfunctional properties. Furthermore, we provided an implementation in a complete MAPE-cycle and showed applicability, effectiveness, and efficiency of our solution by means of a real-world case study.

In future work, we will apply online-learning algorithms to continuously refine the performance-influence models to changing context specifications at runtime. Additionally, we will add temporal constraints between reconfigurations decisions, resource constraints, and reconfiguration costs. Furthermore, we plan to explore new usage scenarios from the communication domain as optimal adaptation of complex event processing engines.

## ACKNOWLEDGMENTS

This work has been funded by the German Research Foundation (DFG) as part of projects A1 and A4 within CRC 1053– MAKI. The authors thank Alexander Grebhahn for his valuable support regarding SPL CONQUEROR.

<sup>3</sup>See <https://dev.kom.e-technik.tu-darmstadt.de/simonstrator/publications/>

## REFERENCES

- [1] Evelyn Martin Lansdowne Beale and John A Tomlin. 1970. Special facilities in a general mathematical programming system for non-convex problems using ordered sets of variables. *OR* 69, 447–454 (1970).
- [2] David Benavides, Pablo Trinidad Martín-Arroyo, and Antonio Ruiz Cortés. 2005. Automated Reasoning on Feature Models. In *Advanced Information Systems Engineering, 17th International Conference, CAiSE 2005, Porto, Portugal, June 13-17, 2005, Proceedings*. 491–503. [https://doi.org/10.1007/11431855\\_34](https://doi.org/10.1007/11431855_34)
- [3] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated Analysis of Feature Models 20 Years later: A Literature Review. *Information Systems* 35, 6 (2010), 615–636. <https://doi.org/10.1016/j.is.2010.01.001>
- [4] Nelly Bencomo, Pete Sawyer, Gordon Blair, and Paul Grace. 2008. Dynamically Adaptive Systems are Product Lines too: Using Model-Driven Techniques to Capture Dynamic Variability of Adaptive Systems. In *Proc. of the Intl. Workshop on Dynamic Software Product Lines (DSPL 2008)*. <https://doi.org/10.1109/SPLC.2008.69>
- [5] Tracy Camp, Jeff Boleng, and Vanessa Davies. 2002. A survey of mobility models for ad hoc network research. *Wireless Communications and Mobile Computing* 2, 5 (2002), 483–502. <https://dx.doi.org/10.1002/wcm.72>
- [6] Rafael Capilla, Jan Bosch, Pablo Trinidad, Antonio Ruiz Cortés, and Mike Hinchey. 2014. An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry. *Journal of Systems and Software* 91 (2014), 3–23. <https://doi.org/10.1016/j.jss.2013.12.038>
- [7] Krzysztof Czarnecki, Thomas Bednash, Peter Unger, and Ulrich W. Eisenacker. 2002. Generative Programming for Embedded Software: An Industrial Experience Report. In *Generative Programming and Component Engineering, ACM SIGPLAN/SIGSOFT Conference, GPCE 2002, Pittsburgh, PA, USA, October 6-8, 2002, Proceedings*. 156–172. [https://doi.org/10.1007/3-540-45821-2\\_10](https://doi.org/10.1007/3-540-45821-2_10)
- [8] Krzysztof Czarnecki and Andrzej Wasowski. 2007. Feature Diagrams and Logics: There and Back Again. In *Software Product Lines, 11th International Conference, SPLC 2007, Kyoto, Japan, September 10-14, 2007, Proceedings*. 23–34. <https://doi.org/10.1109/SPLINE.2007.24>
- [9] Falko Dressler, Margit Mutschlechner, Bijun Li, Rüdiger Kapitza, Simon Ripberger, Christopher Eibel, Benedict Herzog, Timo Hönig, and Wolfgang Schröder-Preikschat. 2016. Monitoring Bats in the Wild: On Using Erasure Codes for Energy-Efficient Wireless Sensor Networks. *ACM Trans. Sen. Netw.* 12, 1, Article 7 (Feb. 2016), 7:1–7:29 pages. <https://doi.org/10.1145/2875426>
- [10] R Jo Duffin. 1970. Linearizing geometric programs. *SIAM review* 12, 2 (1970), 211–227.
- [11] Ahmed Elkhodary, Naeem Eshfahani, and Sam Malek. 2010. FUSION: A Framework for Engineering Self-tuning Self-adaptive Software Systems. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10)*. ACM, 7–16. <https://doi.org/10.1145/1882291.1882296>
- [12] Fabian Fuchs, Markus Völker, and Dorothea Wagner. 2012. *Simulation-Based Analysis of Topology Control Algorithms for Wireless Ad Hoc Networks*. Springer Berlin Heidelberg, 188–202. [https://doi.org/10.1007/978-3-642-34862-4\\_14](https://doi.org/10.1007/978-3-642-34862-4_14)
- [13] Elena Gaura, Michael Allen, Lewis Girod, James Brusey, and Geoffrey Challen. 2016. *Wireless Sensor Networks: Deployments and Design Frameworks* (1 ed.). Springer. <https://doi.org/10.1007/978-1-4419-5834-1>
- [14] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wasowski. 2013. Variability-aware performance prediction: A statistical learning approach. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*. 301–311. <https://doi.org/10.1109/ASE.2013.6693089>
- [15] Svein O. Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. 2008. Dynamic Software Product Lines. *IEEE Computer* 41, 4 (2008), 93–95. <https://doi.org/10.1109/MC.2008.123>
- [16] Svein O. Hallsteinsen, Erlend Stav, Arnor Solberg, and Jacqueline Floch. 2006. Using Product Line Techniques to Build Adaptive Systems. In *Software Product Lines, 10th International Conference, SPLC 2006, Baltimore, Maryland, USA, August 21-24, 2006, Proceedings*. 141–150. <https://doi.org/10.1109/SPLINE.2006.1691586>
- [17] Herman Hartmann and Tim Trew. 2008. Using Feature Diagrams with Context Variability to Model Multiple Product Lines for Software Supply Chains. In *Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings*. 12–21. <https://doi.org/10.1109/SPLC.2008.15>
- [18] IBM Corp. 2015. IBM ILOG CPLEX V12.7.5 User's Manual for CPLEX. <http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/>
- [19] Pooyan Jamshidi, Miguel Velez, Christian Kästner, Norbert Siegmund, and Prasad Kawthekar. 2017. Transfer Learning for Improving Model Predictions in Highly Configurable Software. In *12th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS@ICSE 2017, Buenos Aires, Argentina, May 22-23, 2017*. 31–41. <https://doi.org/10.1109/SEAMS.2017.11>
- [20] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-021. Software Engineering Institute, Carnegie Mellon University. <http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=11231>
- [21] Jeffrey O. Kephart and David M. Chess. 2003. The vision of autonomic computing. *Computer* 36, 1 (2003), 41–50. <https://doi.org/10.1109/MC.2003.1160055>
- [22] Roland Kluge, Michael Stein, Gergely Varró, Andy Schürr, Matthias Hollick, and Max Mühlhäuser. 2017. A Systematic Approach to Constructing Families of Incremental Topology Control Algorithms Using Graph Transformation. *Journal of Software and Systems Modeling (SoSyM)* (2017). <https://doi.org/10.1007/s10270-017-0587-8>
- [23] Christian Krupitzer, Felix Maximilian Roth, Sebastian Vansyckel, and Christian Becker. 2015. Towards Reusability in Autonomic Computing. In *Proc. of the 12th Intl. Conf. on Autonomic Computing (ICAC)*. IEEE, 115–120. <https://doi.org/10.1109/ICAC.2015.21>
- [24] Erhan Leblebici, Anthony Anjorin, and Andy Schürr. 2014. Developing eMoflon with eMoflon. In *Proc. of the Intl. Conference on Model Transformation (ICMT) (LNCS)*, Vol. 8568. Springer, 138–145. [https://doi.org/10.1007/978-3-319-08789-4\\_10](https://doi.org/10.1007/978-3-319-08789-4_10)
- [25] Ning Li, J. C. Hou, and L. Sha. 2005. Design and analysis of an MST-based topology control algorithm. *IEEE Trans. on Wireless Communications* 4, 3 (2005), 1195–1206. <https://doi.org/10.1109/TWC.2005.846971>
- [26] Xiang-Yang Li, Peng-Jun Wan, Yu Wang, and O. Frieder. 2002. Sparse power efficient topology for wireless networks. In *Proc. of the 35th Annual Hawaii Intl. Conference on System Sciences (HICSS)*. IEEE, 3839–3848. <https://doi.org/10.1109/HICSS.2002.994518>
- [27] R Timothy Marler and Jasbir S Arora. 2004. Survey of multi-objective optimization methods for engineering. *Structural and multidisciplinary optimization* 26, 6 (2004), 369–395. <https://doi.org/10.1007/s00158-003-0368-6>
- [28] Marcilio Mendonça, Andrzej Wasowski, and Krzysztof Czarnecki. 2009. SAT-based analysis of feature models is easy. In *Software Product Lines, 13th International Conference, SPLC 2009, San Francisco, California, USA, August 24-28, 2009, Proceedings*. 231–240. <https://doi.org/10.1145/1753235.1753267>
- [29] Kim Mens, Rafael Capilla, Nicolás Cardozo, and Bruno Dumas. 2016. A taxonomy of context-aware software variability approaches. In *Companion Proceedings of the 15th International Conference on Modularity, Málaga, Spain, March 14 - 18, 2016*. 119–124. <https://doi.org/10.1145/2892664.2892684>
- [30] Carlos Andres Parra, Daniel Romero, Sébastien Mosser, Romain Rouvoy, Laurence Duchien, and Lionel Seinturier. 2012. Using constraint-based optimization and variability to support continuous self-adaptation. In *Proceedings of the ACM Symposium on Applied Computing, SAC 2012, Riva, Trento, Italy, March 26-30, 2012*. 486–491. <https://doi.org/10.1145/2245276.2245370>
- [31] Martin Pfannemüller, Christian Krupitzer, Markus Weckesser, and Christian Becker. 2017. A Dynamic Software Product Line Approach for Adaptation Planning in Autonomic Computing Systems. In *2017 IEEE International Conference on Autonomic Computing, ICAC 2017, Columbus, OH, USA, July 17-21, 2017*. 247–254. <https://doi.org/10.1109/ICAC.2017.18>
- [32] Björn Richerzhagen, Dominik Stingl, Julius Rückert, and Ralf Steinmetz. 2015. Simulator: Simulation and Prototyping Platform for Distributed Mobile Applications. In *Proc. of the Intl. Conf. on Simulation Tools and Techniques (SIMUTools)*. ICST, 99–108. <https://doi.org/10.4108/eai.24-8-2015.2261064>
- [33] George F. Riley and Thomas R. Henderson. 2010. The ns-3 Network Simulator. In *Modeling and Tools for Network Simulation*, Klaus Wehrle, Mesut Günes, and James Gross (Eds.). Springer, 15–34.
- [34] Paolo Santi. 2005. Topology Control in Wireless Ad Hoc and Sensor Networks. *ACM computing surveys (CSUR)* 37, 2 (2005), 164–194. <https://doi.org/10.1145/1089733.1089736>
- [35] Amir Molzani Sharifloo, Andreas Metzger, Clément Quinton, Luciano Baresi, and Klaus Pohl. 2016. Learning and evolution in dynamic software product lines. In *Proceedings of the 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS@ICSE 2016, Austin, Texas, USA, May 14-22, 2016*. 158–164. <https://doi.org/10.1145/2897053.2897058>
- [36] Norbert Siegmund, Alexander Grebahn, Sven Apel, and Christian Kästner. 2015. Performance-influence models for highly configurable systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. 284–294. <https://doi.org/10.1145/2786805.2786845>
- [37] Norbert Siegmund, Sergiy S. Kolesnikov, Christian Kästner, Sven Apel, Don S. Batory, Marko Rosenmüller, and Gunter Saake. 2012. Predicting performance via automated feature-interaction detection. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. 167–177. <https://doi.org/10.1109/ICSE.2012.6227196>
- [38] Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. 2012. SPL Conqueror: Toward optimization of non-functional properties in software product lines. *Software Quality Journal* 20, 3 (01 Sep 2012), 487–517. <https://doi.org/10.1007/s11219-011-9152-9>
- [39] Roopak Sinha, Kenneth Johnson, and Radu Calinescu. 2014. A scalable approach for re-configuring evolving industrial control systems. In *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation, ETFA 2014, Barcelona, Spain, September 16-19, 2014*. <https://doi.org/10.1109/ETFA.2014.7005126>
- [40] Niklas Sorensson and Niklas Een. 2005. Minisat v1.13—a sat solver with conflict-clause minimization. *SAT* 2005, 53 (2005), 1–2.



- [41] Gustavo Sousa, Walter Rudametkin, and Laurence Duchien. 2017. Extending Dynamic Software Product Lines with Temporal Constraints. In *12th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS@ICSE 2017, Buenos Aires, Argentina, May 22-23, 2017*. 129–139. <https://doi.org/10.1109/SEAMS.2017.6>
- [42] Michael Stein, Tobias Petry, Immanuel Schweizer, Martina Bachmann, and Max Mühlhäuser. 2016. Topology Control in Wireless Sensor Networks: What Blocks the Breakthrough?. In *Proc. of the Intl. Conf. on Local Computer Networks (LCN)*. IEEE, 389–397. <https://doi.org/10.1109/LCN.2016.67>
- [43] Guodong Sun, Lin Zhao, Zhibo Chen, and Guofu Qiao. 2015. Effective link interference model in topology control of wireless Ad hoc and sensor networks. *Journal of Network and Computer Applications* 52, Supplement C (2015), 69–78. <https://doi.org/10.1016/j.jnca.2015.01.006>
- [44] Markus Weckesser, Malte Lochau, Thomas Schnabel, Björn Richerzhagen, and Andy Schürr. 2016. Mind the Gap! Automated Anomaly Detection for Potentially Unbounded Cardinality-based Feature Models. In *19th Int. Conference on Fundamental Approaches to Software Engineering (FASE)*. 158–175. [https://doi.org/10.1007/978-3-662-49665-7\\_10](https://doi.org/10.1007/978-3-662-49665-7_10)
- [45] H Paul Williams. 2013. *Model Building in Mathematical Programming*. John Wiley & Sons.