A Dynamic Software Product Line Approach for Adaptation Planning in Autonomic Computing Systems

Martin Pfannemüller*, Christian Krupitzer*, Markus Weckesser†, and Christian Becker*
*Chair of Information Systems II, University of Mannheim
Schloss, 68131 Mannheim, Germany
Email: {martin.pfannemueller, christian.krupitzer, christian.becker}@uni-mannheim.de
†Real-Time Systems Lab, Technische Universität Darmstadt
Merckstr. 25, 64283 Darmstadt, Germany
Email: markus.weckesser@es.tu-darmstadt.de

***Abstract*—Modeling the reasoning component of self-adapting systems including its context is a challenging task. Context feature models used in dynamic software product lines help to capture the capabilities of a software as well as the monitored context values. This enables the possibility to add constraints between the context and system features. In this paper, we present an adaptation logic architecture for specifying the knowledge for reasoning in a model-based manner by means of dynamic software product lines. The whole knowledge for reasoning is encapsulated inside a component which enables the reuse of the adaptation logic for various application scenarios. Thus, the system designer only has to specify the adaptation logic's knowledge and implement the according interfaces in the managed resource. We evaluate the adaptation logic using our architecture in a distributed computing scenario.**

I. Introduction

Despite the availability of frameworks and model-based approaches: building context-aware self-adaptive systems is still a challenging task ([1], [2]). With increasing mobility of computing systems, the context of a system gets gradually more important for reasoning about its state. However, in most existing approaches a more explicit integration of the context into the reasoning process is needed [2]. Additionally, for building such a system, the system designer needs to have expertise in self-adaptive systems development as well as domain knowledge of the actual application.

A self-adaptive system "modifies its own behavior in response to changes in its operating environment" ([3], [4]). It consists of an adaptation logic (AL) and the resources managed by it. The AL follows the MAPE principle, hence, monitors the environment and system and may modify the resources' parameters, structure, or both. Multiple so-called self-* properties are the foundation of the adaptation capabilities [3].

In order to model the adaptation decisions, the Software Product Line (SPL) technique can be used. SPLs are used to "design and implement a products family from which individual products can be systematically derived" [5]. The SPL technique uses feature models to specify possible valid product variants at design time that represent a configuration consisting of a set of features. This creates a state space of possible valid configurations. This state space usually is much larger than the number of features [6]. Feature models typically are represented by a tree structure with a hierarchy of features as well as variation points of the whole SPL. The Feature-Oriented Domain Analysis (FODA) approach was the first feature modeling technique [7]. The approach was extended multiple times with new (graphical) elements [8]. Dynamic SPLs (DSPLs) are an extension of the SPL approach introducing the selection and deselection of features at runtime rather than at design time. This enables the expression of adaptation rules for software products at design time, e.g., by adding a model of the software's context ([5], [9]). By defining constraints between context features and system features, mappings between context situations and reconfigurations of the system are specified. Thus, software created on the basis of a DSPL context feature model is capable of changing itself according to the context at runtime.

In this paper, we integrate a DSPL-based context feature model into the knowledge component of an adaptation logic for facilitating the analysis and the planning of reconfigurations. This enables reasoning on the foundation of context situations. The contribution of this paper is twofold: (1) We define an adaptation logic architecture including a knowledge component which is based on context feature modeling. (2) We present a generic approach that uses this architecture and explicitly the knowledge for providing a solver-based adaptation logic without the need to make use case specific changes to its components. Thus, developers only have to define the monitored parameters as well as a context feature model including priorities and costs as part of the knowledge component in order to get a functioning adaptation logic. We evaluate the adaptation logic based on our proposed architecture by managing the Tasklet system for mobile code offloading [10].

The remainder of this paper is structured as follows: Section II presents related work in the domain of DSPLs for adaptive

2474-0756/17 $31.00 © 2017 IEEE
DOI 10.1109/ICAC.2017.18

systems. The subsequent section describes our approach for context feature model-based reasoning. Following, Section IV presents the implementation of our approach using the FESAS framework ([11], [12]). Section V outlines the use case evaluation and Section VI discusses the results of the evaluation. Last, Section VII summarizes this paper and describes future work.

II. Related Work

Hallsteinsen *et al.* introduced the concept of DSPLs in [13]. The authors proposed to shift the concepts of SPLs from design time to runtime for offering adaptation variability. Since then, researchers addressed different aspects of DSPLs. Good overviews over the literature are provided in [14] and [15]. In the following, we focus on the most prominent approaches of DSPLs.

Service-Oriented Dynamic Software Product Lines [16]: The authors combine the Common Variability Language (CVL) with the Business Process Execution Language (BPEL) and aspect-oriented programming. The variability designer uses CVL, e.g., using an existing Eclipse plugin to model the changed configuration. This new configuration leads to a change request. For the execution, the tool DyBPEL augmenting the ActiveBPEL execution engine with aspect-oriented variability. ActiveBPEL is an execution engine which allows running business processes defined using BPEL. As part of this DyBPEL engine, a coordination component receives change requests from the variability designer. It triggers a change of the execution inside the embedded ActiveBPEL engine as well as a runtime modifier migrating running processes.

Genie [17]: The approach supports the development and modeling of reconfigurable component-based systems. Genie uses two model structures for representing the system: a context variability model and a structural variability model. These dimensions are connected for representing the reconfiguration behavior. The environment states are represented as state diagram with transitions between the states.

Applying Software Product Lines to Build Autonomic Pervasive Systems [18]: Cetina *et al.* developed a DSPL approach in the domain of pervasive computing. They use a model-driven methodology for specifying the features and the behavior. Features are modeled using the extended feature modeling notation of [19]. The system structure, as well as the configuration behavior, is described using the PervML modeling language. They developed an additional mapping between the features of the feature model and the PervML elements which describes the adaptation behavior and provides self-healing capabilities by specifying fallback mechanisms.

MADAM [20]: The system is represented as a component model where components conforming to a matching interface can be easily replaced. Additionally, MADAM supports parametrization of the components. It works utility-based which means that the main goal of the developed middleware is the maximization of the utility. The utility depends on the current context situation that gets evaluated continuously. According to the context change, MADAM activates the components with the highest expected utility.

REPFLC [21]: The approach integrates a three phases lifecycle: product family engineering, target system configuration, and target system reconfiguration. Product family engineering includes the creation of a product family architecture consisting of components and connections as well as the specification of variation points. Reconfiguration patterns for the runtime adaptation specify possible transitions between configurations. Additionally, a state model describes the changes of an adaptation and a scenario model defines certain conditions when an adaptation should be triggered. During the target system configuration, the system gets actually deployed. Finally, the target reconfiguration phase uses the models specified at design time to support runtime adaptations.

DiVA [22]: This approach is based on four metamodels. They consist of (i) a DSPL model, (ii) a context model, (iii) a reasoning model, and (iv) an architecture model. The DSPL model represents a standard feature model. The context model consists of single variables needed for monitoring at runtime. Reasoning means to connect the two former models for triggering adaptations, e.g., with ECA rules. The architecture model can be any architecture model such as a standard UML model. These models are used in a three layer architecture. The bottom layer contains the AL while the top layer contains the actual AL. The middle layer connects both of them. A reasoner picks the best configuration based on the context information. A consistency checker for the models used at runtime concludes the approach.

Context awareness for dynamic service-oriented product lines [23]: Parra *et al.* use the context-sensing middleware COSMOS which provides so-called context nodes. A context node has always the same interface to the application and provides the context information of one sensor. This approach abstracts the sensors from the application. A context model is the basis for storing the current context. Each so-called context-aware asset is, e.g., defined by some value that should be observed as well as by the thresholds of the value and the changes that should be implemented in each case. For representing the system variability, a standard feature model is used. Changes in the context-aware assets applied by a context manager trigger changes in the architecture.

As the extended comparison in [24] shows, all methods work component-based and autonomously. Most of the approaches apply static goal evolution and focus on self-configuration. The majority of the approaches react on changes in the context. Most approaches require that the developer has to learn a new modeling technique which is not applied in general. Also, some approaches target a specific application domain which might make them not easily applicable in other domains. In this paper, we propose a more generic DSPL approach for adaptation planning. The approach integrates a separated context model and uses cross-tree constraints between the context and system features. In the following section, we present the architecture of our approach.

III. System Model

In this section, we present our context feature model-based architecture for the AL. First, we introduce the modeling approach and its representation for the planner component. Second, we present our architectural design integrated into the MAPE-K cycle.

A. Context-Aware Feature Modeling Approach

This work uses the feature modeling approach introduced in [9]. They augmented generic feature model diagrams with a context branch in addition to the system features. This explains the name context feature model (CFM) of this model type. Moreover, the approach uses feature attributes and feature instance cardinalities as well as group type cardinalities for specifying constraints in the model. Cross-tree constraints are used between context features, feature attributes, and system features for specifying the reconfiguration behavior. They can either require or exclude a feature. At runtime, the selection of context feature attributes represents the current state of the running software.

For finding valid configurations at runtime, the context feature model is converted to a representation interpretable by a satisfiability problem (SAT) solver. Each feature represents a literal which can either have the value true or false. Cardinalities and cross-tree constraints can also be translated into a boolean representation. Since a SAT solver only works with boolean problem representations, the context feature attribute value ranges are modeled as enumerations either being true or false. The complete context feature model is translated into DIMACS conjunctive normal form (CNF) [25]. This is the de facto standard of representing SAT problems for solvers. The solver itself is integrated into the planner component as described in detail in the following section.

B. Architecture of the AL

As a starting point, the idea is to augment a MAPE-K cycle based AL with a CFM inside the knowledge component. Furthermore, the knowledge contains rules for relating raw sensor data to context feature attributes. Also, there is a SAT mapping which relates every feature and feature attribute to its literal representation for the SAT solver as part of the knowledge. This mapping is built directly by the knowledge component at the start of the system. This enables the knowledge component to return the corresponding literal given a feature or feature attribute. Additionally, priorities and costs are used for conflict resolution and the selection of one configuration when multiple configurations are valid. All children of one feature form a feature group. Priorities and costs are present for all system features which are part of a feature group. A priority is represented as a number stating the priority of a system feature inside its feature group. The lower the number the higher the priority. A cost value referring to a system feature states the estimated cost to implement exactly this system feature in comparison to other system features of the same feature group. Typically, the value range of both values is between one and the number of features in the group.

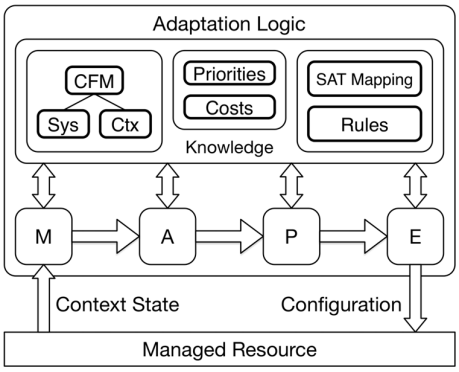

Figure 1. MAPE-K cycle extended by a DSPL context feature model (CFM) and additional information for adaptation planning. Sys = System Features Model, Ctx = Context Features.

Figure 1 shows an overview of the AL's architecture. As usual for a MAPE-K-based approach, the monitoring element gets raw data from the managed resource. In this case the data coming from the managed resource is context information. This data consists of system data and external context information related to the managed resource.

The monitoring component receives the data, prepares it and passes it to the analyzer component. The preparation includes interpretation of serialized input like an XML or JSON string in order to create plain data objects for working with the sensor data. The resulting data objects can be used more easily by the analyzer component than a raw string. The AL is able to receive and process partial sensor data until the managed resource sends a message indicating the end of data. Thus, the monitor gets sensor information not as one package but in fragments.

The analyzer creates the average of all entries in order to create one single sensor value representing the average system state. Each average system value is used to map it's value to actual context feature attributes representing the context state of the system. With minor customization, developers can specify other aggregation functions rather than the average. In this case, after the creation of the average values, the rules representing the relationships of context information, their names, and context feature attributes are used. Matching the actual values to context feature attributes requires rules stating the value range of each context feature attribute. Thus, it is possible to match the average values to actual context feature attributes. The approach supports partial knowledge, hence, not for all context feature attributes data must be present. Even without full knowledge, the system is capable of finding a valid configuration. The resulting attributes which are selected according to the context information are forwarded to the planner component.

The workflow of the planner is shown in Figure 2. As the planner mainly works with a SAT solver, the result of the mapping process at the beginning is a logical representation of the context information in conjunctive normal form (CNF). As already stated, DIMACS CNF is used here

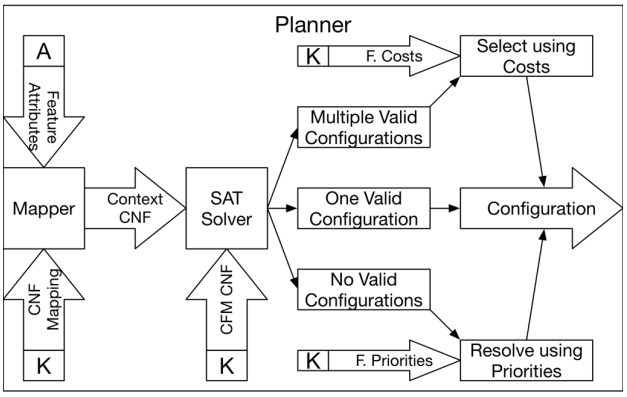

Figure 2. Workflow of the planner component. CFM = context feature model, CNF = conjunctive normal form, A = Analyzer, F = Feature, K = Knowledge.

as logical representation (see [25]). The mapper uses the SAT CNF mappings of the knowledge component. These mappings state which feature or feature attribute is mapped to which literal for representation inside the solver. Each context feature attribute generated by the analyzer is mapped to its literal representation resulting in logical clauses in CNF. The result is a DIMACS CNF representation of the entire context information. The planner then uses the knowledge component to access additional system information. The CFM - which is available in CNF representation as well - is used in conjunction with the CNF representation of the context information as input for a SAT solver. The solver's task is to determine if valid configurations exist and to output all possible configurations. If there is only one valid configuration, the planner is finished as there are no other configuration options it can choose from. In case of multiple possible configurations, the additional cost information residing in the knowledge component is used. The costs contain a numeric cost value for each system feature as part of a feature group inside the CFM. Using this, the planner then selects the configuration with the lowest cost. In case no valid configuration is found, the planner has to solve the conflict somehow. A conflict occurs if unforeseen context situations arise which were not anticipated at design time. Our system can also handle unknown or conflicting context states at runtime. If a conflict occurs, there may be two or more system features in conflict. In this case, the priority information is used. It determines the priority of system features in relation to other system features in the same feature group. The planner selects the feature with the highest priority for each conflicting feature group. One example is a smart home system. One context feature requires to turn on the air conditioning when it is hot in the room. Another context feature requires the system to stop the air conditioning as it is in the middle of the night and someone sleeps in the same room. In this case, the user may have specified the priorities in the first place resulting in his desired behavior in this situation. Thus, if the user always wants to sleep in a cold room, he could set a higher priority to the feature turning on the air conditioning. The result of the planner is a complete list of system features the managed resource should activate. The selected configuration is sent to the execution component which forwards the configuration to the corresponding managed resource. This ends one complete cycle through the AL. The next chapter shows implementation details of the AL which is based on this architecture.

IV. Implementation

This section describes the implementation of a generic AL which uses our method for planning reconfigurations based on context feature models. The implementation is based on the approach presented in the previous section. It is generic and reusable as developers just have to define the corresponding elements of the system knowledge without the need for changes in any other AL component. We used Java and the FESAS framework for implementing the AL as it abstracts from issues as communication within the AL and, hence, fastens development as developers may concentrate on the logic of their SAS. For further information regarding the FESAS framework, the interested reader is referred to [11] and [12].

For illustration purposes, an entire run through the AL is described using the data center use case presented in [11]. It describes a self-managing data center that starts servers given a high workload. Accordingly, it should stop or keep the number of servers in low workload situations. Also, it is possible to redistribute virtual machines over all physical servers for better resource utilization. Figure 3 shows the big picture of the data flow through the MAPE components including the context feature model of the example which is part of the knowledge component. For illustration, the monitor only observes the workload of the servers in this simplified example. Furthermore, the system features consist of a startup policy and a keep policy for the server management. Additionally, the VM management possibilities include a redistribution and a stay policy. In this example, one AL manages three data center areas. The monitor receives sensor information about the workload from each of the three data center areas followed by a keyword for stating that the monitor should stop monitoring and forward the average of the received values to the analyzer. The analyzer selects the corresponding feature attribute item (FAI) component. One FAI represents a feature attribute in the implementation. Based on this input the planner selects the startup policy and the redistribution policy. The result is sent as a concatenated string to the executor which forwards it to a corresponding effector interface. In the following, implementation details with reference to the example of all MAPE-K components are explained.

The knowledge contains all elements that are needed for reasoning in the AL. The context feature model is only one part of the knowledge besides the (feature attribute) rules, the priorities, and the costs. Features can be either system or context features. Each feature can have two types of cardinalities: a feature instance cardinality and a group type cardinality. The feature instance cardinality specifies how many instances of a feature are allowed to be present in the

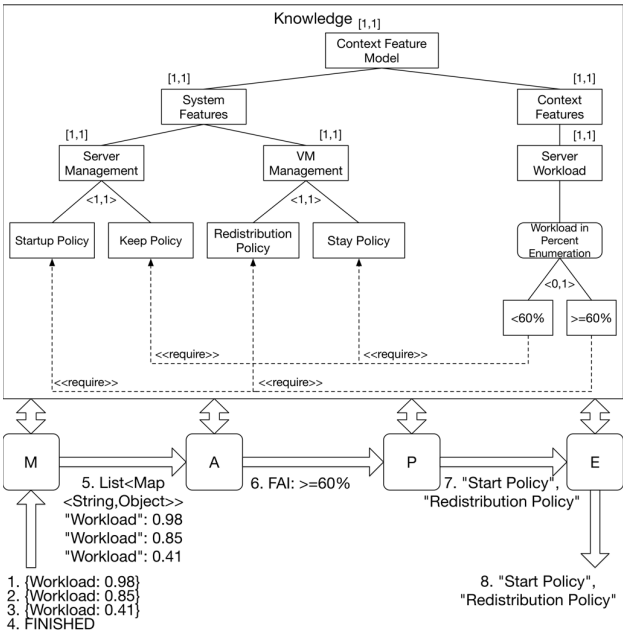

Figure 3. Complete data flow between the MAPE components. The knowledge component only shows the CFM. FAI = `FeatureAttributeItem`, WL = Workload. Feature instance cardinalities are denoted with square brackets. Group type cardinalities are denoted with angle brackets.

system at runtime. The group type cardinality states the type for the feature group of the feature's children. For example, a feature group denoted with the cardinalities <1,1> means exactly one child can be selected at the same time. According to the extended feature model approach, features can also have attributes. Feature attribute rules represent the rules for matching context sensor values to actual feature attributes and feature attribute items. For each feature attribute type, a corresponding rule type is required. A rule specifies the name for the sensor input matching it to an attribute. The rules have to be specified by a developer at design time. Additionally, it provides a matching method for determining the attribute item that is represented by some context value. For the example in Figure 3, the attribute workload would have the rules: (1) (Workload<0.6) mapped to the "<60%" FAI and (2) (Workload>=0.6) mapped to the ">=60%" FAI. The complete metamodel of the knowledge component can be found in [24].

The monitoring component gets multiple raw strings as sensor input. It converts the raw JSON data to Java objects and adds it to an array list. In the example in Figure 3, three JSON strings are stored in a `List<Map<String,Object>>` object. Each string represents the current status of the workload of one data center. Passing this array list to the analyzing component triggers the analyzer.

The analyzing component analyzes the values and entries. Referring to the example in Figure 3, the analyzer creates the average workload. This is done for every variable over all entries of the list. The resulting value is mapped to the actual feature attributes items representing the context situation of these averaged values. In the example, only the FAI *>=60%* is selected. Then the analyzer sends the information about the FAIs to the planning component.

In the previous section, we described the general planning approach. For actually implementing the planner component, we use *Sat4J* as SAT solver [26]. First, using the SAT mapping, the planner maps context feature attributes to their literal representation. Next, it checks if there is a model using the given context information. At this point, the solver already has loaded the CNF of the CFM. If there is no model, the conflicting features are determined. The conflicting features are passed to the `PriorityConflictSolver` which identifies the system features responsible for the conflict. For each feature group with conflicts, the feature with the highest priority is selected. Adding the resolved feature selection as well as the remaining conflict-free context information to the solver ends the conflict resolution. If the model is directly free of conflicts, the context information is added to the solver. If multiple models are valid, the `CostConfigurationSelector` uses the costs assigned to the system features. For every feature group, the feature with the lowest cost is selected. The planner sends a list with all selected system features to the executor.

The execution only forwards the feature selection towards the managed resource. As shown in Figure 3, the feature selection is a string with the system feature names separated by a comma. The managed resource maps the system feature names to actual actions.

In our implementation, the connection between AL and managed resource is implemented using socket connections. The socket connections are encapsulated inside sensor and effector components connected to the monitor and execution component respectively. The message from the execution component to the effector only contain the names of the selected system features as strings. Hence, the managed resource must interpret the result of the AL and start and stop policies accordingly on its own. Using the other socket, the raw context information is transported in the JSON format to the sensor component. It is the responsibility of the managed resource to provide the context information in the correct JSON format.

V. Evaluation

This section outlines the evaluation of the system. First, we present the use case of the evaluation. Second, we pose the evaluation questions and describe the corresponding evaluation scenarios. Third, we describe the results of the evaluation.

A. Use Case Description

Using our approach for planning based on CFM, we implemented a system for managing the Tasklet system [10]. The idea of the Tasklet system is to provide a middleware for distributed computing on heterogeneous devices. Therefore, three entities are available: resource providers, resource consumers, and resource brokers. Providing resources

means to offer a Tasklet virtual machine (TVM). Resource consumers send their code to providers for remote execution. Additionally, local execution is possible. Each resource provider registers at a broker while brokers themselves form a peer-to-peer overlay network. Consumers send resource requests to a broker, which then searches for a suitable resource provider. A consumer may specify different levels of non-functional requirements called Quality of Computation for a Tasklet, e.g., reliability, speed, or security. This may limit the set of possible providers. The broker returns the information for connecting to a provider. The consumer uses this knowledge to directly send a Tasklet to this provider. Each entity in the network runs the Tasklet middleware that handles the construction of Tasklets, their execution, and distribution. An overview of an example overlay network topology is shown in Figure 4. For more details on the Tasklet system, the interested reader is referred to [10].

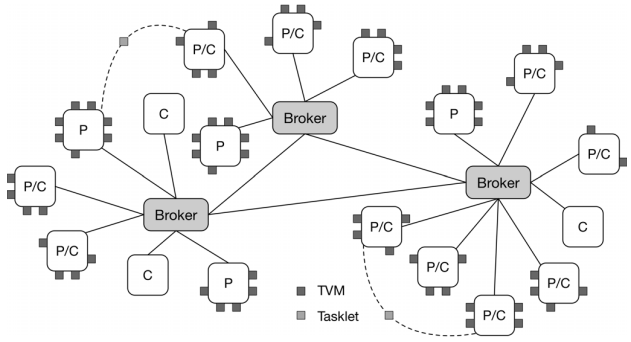

Figure 4. Schema of a Tasklet network topology. P = Provider, C = Consumer, TVM = Tasklet Virtual Machine. [10]

We simulated the Tasklet system using the simulated system presented in [27]. Additionally, we added a broker manager entity. It stores a list of all brokers, monitors them and adapts their behavior by changing their configurations according to the results of the AL. Besides changing the configuration of brokers, it can start and stop brokers, as well as redistribute entities connected to them. As the simulator is discrete, we synchronized the simulation with the AL through the broker manager. Hence, the broker manager transmits all monitored data at the end of a simulation step to the AL and pauses the simulation. After each MAPE activity performed its task, the broker manager adapts the simulated system in case of a new system configuration. Afterward, it starts the next simulation step. In case of a real system, adaptations would occur continuously. Thus, the simulated system does not represent this aspect of the real world. As mentioned in the previous section, broker manager and AL are connected via sockets in our approach. A more detailed description of the simulation environment can be found in [24].

In order to adapt the settings of the brokers through the AL, the first step is to identify system features for the management system as well as context information that is needed to plan the selection of the system features. One cross tree constraint in it is concerned with the startup of new brokers if the load of the brokers exceeds 60%. Accordingly, brokers are stopped with a load below 60%. Additionally, a feature for latency optimized placement of nodes to near brokers is selected in case of an overall high latency between nodes and their corresponding brokers. This should result in lower load per broker as well as lower latencies. The complete CFM can be found in [24]. No customizations on the MAPE components including the sensor and effector components were necessary. The managed resource simply had to support the specified socket connections for data exchange.

B. Evaluation Settings

To evaluate the performance of our system, we define two evaluation questions (EQ):

(EQ1) Does the AL improve the system performance and how does the amount of participants in the network influence it?
(EQ2) How does the amount of system and context features influence the performance of the adaptation planning approach?

These questions result in two evaluation scenarios.

Scenario 1: For identifying improvements of the AL, we compare multiple run configurations with and without the AL. In the first setting 1 000 Providers, 500 consumers, and two brokers are the start configuration. In the following setting we use 5 000 providers, 2 500 consumers, and 10 brokers. Additionally, the system behavior with different ratios between providers and consumers is evaluated. Thus, we also evaluated the 1:1 and 1:2 ratios resulting in 5 000 providers and consumers in setting 3, and 2 500 providers and 5 000 consumers in setting 4. The evaluation always stops after 25 000 finished Tasklets.

Scenario 2: As the CFM of the use case has 20 system features and 5 context features only, for this scenario, we implemented a `KnowledgeFactory` class for creating random models. It creates random system features, context features, feature attribute items, constraints between them, costs, priorities, and feature attribute rules. Additionally, we implemented a `ModelTester` class for testing the AL with data generated by the `KnowledgeFactory`. Since the value range of the context feature attributes is fixed, random values fitting the value range are created for monitoring. We compare the time from sending the random values to the monitor until a feature selection is generated for evaluation. According to the model size of the use case, for the baseline the setting 20/5 (system features/context features) is used for the first model creation process. We did not use the use case model here for having a comparison of randomized models only. Then the multiples 100/25, 1 000/250 and 10 000/2 500 are used. For each configuration, 10 knowledge input files are created. This results in 40 models. Each model is executed in a test environment 10 times resulting in 400 results. The complete execution time per run is gathered.

C. Evaluation Results

This section presents the aggregated results of the evaluation. An interpretation and a cross-setting discussion of the results follows in the subsequent section.

Scenario 1: We measured two variables in the Tasklet system: the average latency (in milliseconds) and the average load of the brokers in the Tasklet system. For both values, the AL triggers a system feature adaptation if they exceed a threshold. We tested different distributions between providers and consumers. As a result, the simulated system using the AL has on average a 43% lower latency and the brokers have on average 45% less load than the simulated system without the AL. Table I shows the measured values per run. The 2 500/5 000 setting was not able to finish without the AL. Every run ended in an OutOfMemory exception. We also tested it on an r4.2xlarge instance from Amazon EC2 with 61 GB of ram. Even on this machine, we were not able to finish a single run in this configuration. One reason is that the overall simulated load in the Tasklet system is very high. The ratio between providers and consumers in this setting increases the load even further. This results in a very high number of steps for finishing the predefined 25 000 Tasklets. Thus, for finishing one run the JVM has to have all simulation data present at the whole simulation time. In any case, it shows that the AL improves the performance also in this setting. The table shows that the other results are very similar in all four settings. It also indicates that the AL is able to fulfill the adaptation goals in all settings.

Table I
AGGREGATED LATENCY AND LOAD ON THE BROKERS FOR THE FIRST EVALUATION QUESTION. AV = AVERAGE.

Setting	Av latency no AL	Av latency AL	Av load no AL	Av load AL
1 000/500	18.38 ms	10.67 ms	100 %	54.70 %
5 000/2 500	18.35 ms	10.52 ms	100 %	55.36 %
5 000/5 000	19.12 ms	11.65 ms	100 %	55.27 %
2 500/5 000	-	11.22 ms	-	55.29 %

Scenario 2: For answering the second evaluation question, we compared the number of FAIs and the execution time of the AL in four settings. All results are aggregated and can be seen in Table II. In the three smaller settings the AL found a configuration in significantly less than a second. In the largest setting, the AL was able to find a valid configuration in 8.7 seconds on average.

VI. DISCUSSION

This section discusses the evaluation presented in the previous section. Concerning the first EQ, the AL reduces the average latency as well as the average load of the brokers in all settings. In the first three settings, the latency could be reduced on average by 41.3 %. Respectively the load could be reduced by 44.89 % on average. The most interesting change is the number of simulation steps needed by the simulated systems in the four settings. Table III shows them including the improvement between the aggregated values of the runs without and with the AL. Again, there are no values for the last setting. In the very small first setting, the overhead of the AL including the policies inside the broker management system results in a worse performance using the AL. The larger the setting gets, the higher the improvement regarding the number of saved simulation steps is. The third setting, which is the largest setting regarding the number of the entities, even needs 50 % fewer simulation steps employing the AL.

Table III
AGGREGATED RESULTS OF THE AVERAGE STEPS NEEDED FOR THE FIRST EVALUATION QUESTION.

Setting	Steps no AL	Steps AL	Change
1 000/500	967,67	1 288,03	+33 %
5 000/2 500	1 599,87	1 244,70	-22 %
5 000/5 000	2 205,33	1 110,07	-50 %
2 500/5 000	-	1 301,57	-

The second evaluation question aims to determine if the AL is capable in creating valid configurations quickly for large models. Hence, it mainly focusses on measuring the performance of the SAT solver. The model testing approach showed that even with 10 000 system features, 2 500 context features with up to 9 743 feature attribute items in the model, our approach is able to create an adaptation result within 8.7 seconds on average. Systems with the requirement of low reaction times may need to run the planning on a fast cloud server rather than executing it locally. For comparison it may be possible to integrate additional SAT solvers in future work.

One possibility for general improvement concerns the decision to create an average of all context information. This may lead to decreased system performance at certain entities in the system as the averaged values could hide local problems. Thus, one idea is to add entity-based planning instead of

Table II
MODEL TESTING RESULTS OF SECOND EVALUATION QUESTION: AGGREGATED NUMBER OF FEATURE ATTRIBUTE ITEMS (FAI) AND AGGREGATED RUNTIME IN MS. ST D = STANDARD DEVIATION, AV = AVERAGE.

Features		FAI				Execution time ms			
System Features	Context Features	Min FAI	Max FAI	St D FAI	Av FAI	Min	Max	St D	Av
20	5	10	19	2,47	13,90	16	441	74,88	55,48
100	25	78	90	3,65	83	44	1 043	108,36	75,71
1 000	250	835	892	18,16	871,50	279	543	57,83	350,53
10 000	2 500	7 090	9 743	454,32	7 690,60	8 672	8 834	55,98	8 737,90

an overall system planning. This would result in different configurations for each broker in our use case.

VII. Conclusion

This paper shows that our CFM AL approach is able to fulfill goals stated in a context feature model. Additionally, it enables the reuse of the complete AL without any changes by only exchanging the knowledge inside the knowledge component. For evaluation, we used an implementation in Java using FESAS. The Tasklet distributed computing system is the use case for the quantitative evaluation of our approach. The evaluation shows multiple properties of the developed AL. It shows that the overhead of the AL leads to bad results in small Tasklet settings. The reason for this is that the improvements in such a setting do not compensate for the performance degradations of the overhead. However, the larger Tasklet settings have finished significantly faster employing the AL. Concerning the 2 500/5 000 setting, it was not even possible to get any results without the AL.

For future work, other solvers, as well as entity-based planning, may be added to our approach. Multi-core support may also increase the performance significantly. At the moment, the AL, as well as the simulated Tasklet system, are not optimized for multiple CPU cores. Hence, only one core was used when the evaluation was run. In the current stage, the developer has to (1) define the CFM, (2) set priorities and costs to system features of feature groups and (3) define mappings from raw context information to context features. This has to happen manually at the moment. It may be possible to provide a developer toolset for easily specifying the complete knowledge of the adaptation logic in an intuitive way in the future. Thus, software engineers who are familiar with DSPL context feature models would be able to specify the behavior of the AL without changing the components of the AL itself.

Acknowledgment

This work has been co-funded by the German Research Foundation (DFG) as part of project A4 within the Collaborative Research Center (CRC) 1053 – MAKI. The authors would like to thank Janick Edinger and Dominik Schäfer for their valuable contribution of the use case.

References

[1] J. Floch, C. Frà, R. Fricke, K. Geihs, M. Wagner, J. Lorenzo, E. Soladana, S. Mehlhase, N. Paspallis, H. Rahnama, P. Ruiz, and U. Scholz, "Playing MUSIC - building context-aware and self-adaptive mobile applications," *Software: Practice and Experience*, vol. 43, no. 3, pp. 359–388, 2013.

[2] C. Krupitzer, F. M. Roth, S. VanSyckel, G. Schiele, and C. Becker, "A survey on engineering approaches for self-adaptive systems," *Pervasive and Mobile Computing*, vol. 17, pp. 184–206, 2015.

[3] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[4] P. Oreizy, M. Gorlick, R. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. Wolf, "An architecture-based approach to self-adaptive software," *Intelligent Systems and their Applications, IEEE*, vol. Vol. 14, 1999.

[5] M. Acher, P. Collet, F. Fleurey, P. Lahire, S. Moisan, and J. P. Rigault, "Modeling context and dynamic adaptations with feature models," in *CEUR Workshop Proceedings*, vol. 509, 2009, pp. 89–98.

[6] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay, "Symbolic model checking of software product lines," *Proceeding of the 33rd international conference on Software engineering - ICSE '11*, p. 321, 2011.

[7] K. C. Kang, S. G. Cohen, J. a. Hess, W. E. Novak, and a. S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," *Distribution*, vol. 17, no. November, p. 161, 1990.

[8] L. Chen, M. Ali Babar, and N. Ali, "Variability management in software product lines: a systematic review," *Proceedings of the 13th International Software Product Line Conference*, pp. 81–90, 2009.

[9] K. Saller, M. Lochau, and I. Reimund, "Context-aware DSPLs: model-based runtime adaptation for resource-constrained systems," 2013, pp. 106–113.

[10] D. Schäfer, J. Edinger, J. M. Paluska, S. VanSyckel, and C. Becker, "Tasklets: "Better than Best-Effort" Computing," in *Proccedings of the International Conference on Computer Communication and Networks*. IEEE, 2016, pp. 1–11.

[11] C. Krupitzer, F. M. Roth, S. Vansyckel, and C. Becker, "Towards Reusability in Autonomic Computing," in *Proceedings of the 12th International Conference on Autonomic Computing (ICAC)*, 2015, pp. 115–120.

[12] C. Krupitzer, F. M. Roth, C. Becker, M. Weckesser, M. Lochau, and A. Schürr, "FESAS IDE: An Integrated Development Environment for Autonomic Computing," in *Proceedings of the 13th International Conference on Autonomic Computing (ICAC)*, 2016, pp. 15–24.

[13] S. Hallsteinsen, M. Hinchey, Sooyong Park, and K. Schmid, "Dynamic Software Product Lines," *IEEE Computer*, vol. 41, no. 4, pp. 93–95, 2008.

[14] M. Bashari, E. Bagheri, and W. Du, "Dynamic software product line engineering: A reference framework," *International Journal of Software Engineering and Knowledge Engineering*, vol. 27, no. 2, pp. 191–234, 2017.

[15] J. R. F. Da Silva, F. A. P. Da Silva, L. M. Do Nascimento, D. A. O. Martins, and V. C. Garcia, "The dynamic aspects of product derivation in DSPL: A systematic literature review," 2013, pp. 466–473.

[16] L. Baresi, S. Guinea, P. Liliana, M. Hinchey, S. Park, and K. Schmid, "Service-Oriented Dynamic Software Product Lines," *IEEE Computer*, vol. 45, no. 10, pp. 42–48, 2012.

[17] N. Bencomo, P. Grace, C. Flores, D. Hughes, and G. Blair, "Genie: Supporting the Model Driven Development of Reflective, Component-based Adaptive Systems," 2008, pp. 811–814.

[18] C. Cetina, J. Fons, and V. Pelechano, "Applying software product lines to build autonomic pervasive systems," no. ii, 2008, pp. 117–126.

[19] D. Benavides, P. Trinidad, and A. Ruiz-Cortés, "Automated Reasoning on Feature Models," *LNCS Advanced Information Systems Engineering 17th International Conference CAiSE 2005*, vol. 01, pp. 491–503, 2005.

[20] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjørven, "Using architecture models for runtime adaptability," *IEEE Software*, vol. 23, no. 2, pp. 62–70, 2006.

[21] H. Gomaa and M. Hussein, "Dynamic Software Reconfiguration in Software Product Families," *Software Product-Family Engineering*, vol. 3014, no. iii, pp. 435–444, 2004.

[22] B. Morin, O. Barais, J.-M. Jezequel, F. Fleurey, and A. Solberg, "Models@ Run.time to Support Dynamic Adaptation," *IEEE Computer*, vol. 42, no. 10, pp. 44–51, 2009.

[23] C. Parra, X. Blanc, and L. Duchien, "Context awareness for dynamic service-oriented product lines," 2009, pp. 131–140.

[24] M. Pfannemüller, "A Dynamic Software Product Line Approach for Planning and Execution of Reconfigurations in Self-Adaptive Systems," Master's thesis, University of Mannheim, Germany, 2017, https://ub-madoc.bib.uni-mannheim.de/41782/.

[25] Satcompetition.org, "Cnf file format," 2009, accessed: 28.02.2017. [Online]. Available: http://www.satcompetition.org/2009/format-benchmarks2009.html

[26] D. Le Berre and A. Parrain, "The Sat4j library, release 2.2 system description," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 7, no. 2010, pp. 59–64, 2010.

[27] J. Edinger, D. Schäfer, C. Krupitzer, V. Raychoudhury, and C. Becker, "Fault-avoidance strategies for context-aware schedulers in pervasive computing systems," in *2017 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, 2017, pp. 79–88.