UNIVERSITÄT MANNHEIM

A Dynamic Software Product Line Approach for Planning and Execution of Reconfigurations in Self-Adaptive Systems

Masterarbeit
von

Martin Pfannemüller
Matrikelnummer 1341300

13. Januar 2017

Vorgelegt am Lehrstuhl für Wirtschaftsinformatik II
Universität Mannheim
Gutachter: Prof. Dr. Christian Becker
Betreuer: Christian Krupitzer

Eidesstattliche Erklärung:
Hiermit versichere ich, dass diese Arbeit von mir persönlich verfasst wurde und dass ich keinerlei fremde Hilfe in Anspruch genommen habe. Ebenso versichere ich, dass diese Arbeit oder Teile daraus weder von mir selbst noch von anderen als Leistungsnachweise andernorts eingereicht wurden. Wörtliche oder sinngemäße Übernahmen aus anderen Schriften und Veröffentlichungen in gedruckter oder elektronischer Form sind gekennzeichnet. Sämtliche Sekundärliteratur und sonstige Quellen sind nachgewiesen und in der Bibliographie aufgeführt. Das Gleiche gilt für graphische Darstellungen und Bilder sowie für alle Internet-Quellen.
Ich bin ferner damit einverstanden, dass meine Arbeit zum Zwecke eines Plagiatsabgleichs in elektronischer Form anonymisiert versendet und gespeichert werden kann. Mir ist bekannt, dass von der Korrektur der Arbeit abgesehen werden kann, wenn diese Erklärung nicht erteilt wird.

Mannheim, 13. Januar 2017 Martin Pfannemüller

Contents

List of Figures

List of Tables

List of Code Listings

List of Abbreviations

AL	Adaptation Logic
BPEL	Business Process Execution Language
BPMN	Business Process Model and Notation
CFM	Context Feature Model
CNF	Conjunctive Normal Form
CSP	Constraint Satisfaction Problem
CSV	Comma-Separated Values
CVL	Common Variability Language
DiVA	Dynamic Variability in complex, Adaptive system
DSPL	Dynamic Software Product Line
DTO	Data Transfer Object
ECA	Event Condition Action
FAI	Feature Attribute Item
FESAS	Framework for Engineering Self-Adaptive Systems
FODA	Feature-Oriented Domain Analysis
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
MADAM	Mobility and Adaptation-enabling Middleware
MAPE(-K)	Monitoring Analyzing Planning Executing (-Knowledge)
OWL	Web Ontology Language
PLE	Product Line Engineering
REPFLC	Reconfigurable Evolutionary Product Family Lifecycle
SAS	Self-Adaptive System
SAT	Satisfiability
SCA	Service Component Architecture
SEI	Software Engineering Institute

SMT	Satisfiability Modulo Theory
SPARQL	SPARQL Protocol and RDF Query Language
SPL	Software Product Line
TVM	Tasklet Virtual Machine
UML	Unified Modeling Language
WL	Workload
XML	Extensible Markup Language

1. Introduction

Today's world consists of an increasing number of information systems that help us handling the growing amount of information. The increasing number of interconnected systems results in higher management efforts and complexity. According to Laddaga et al., this complexity arises from the "growth in problem size", the "increased hardware capacity", real-time, and enterprise requirements [Lad01, p. 1]. All these aspects can be found in many scenarios such as data centers. They constantly grow in size and must meet gradually more and stricter conditions in terms of manageability and scalability. As one example, they have to be easily controllable and they must meet dynamic demands at runtime. Thus, the management effort is very high for data centers today. The high effort requires innovative software solutions for better management.

Self-adaptive software is able to cope with this growing complexity in an autonomous way. According to Oreizy et al. self-adaptive software "modifies its own behavior in response to changes in its operating environment" [OGT+99, p. 55]. A self-adaptive system (SAS) consists of an adaptation part, the so-called adaptation logic, and the resources managed by the adaptation logic. It monitors the changing environment and may modify the parameters, the structure of the managed resource, or both as reaction. This changes the behavior of the managed resources and therefore the output of the software. Multiple so-called self-* properties are the foundation of the adaptation capabilities.

In order to model software capabilities such as these in a concrete way 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" [ACF+09, p. 1]. The SPL approach usually uses feature models to specify possible valid product variants. A valid product variant is represented by a configuration consisting of a set of features. This creates a state space of possible states with each one representing a valid configuration. This state space usually is much larger than the number of features [CHSL11]. Different feature

allocations for the variants of the product line distinguish the products from each other. Dynamic SPLs are an extension to this approach introducing the allocation of features at runtime rather than at design time. This enables the dynamic SPLs to express adaptation rules for software products at design time, e.g., by adding a model of the context the software is running in [ACF+09, SLR13]. By defining constraints between context states and system features, mappings between context situations and reconfigurations of the system are specified. Thus, software created on the basis of a DSPL is capable of changing itself according to the context at runtime.

1.1. Objective and Approach

The objective is to identify a standardized and reusable process of creating adaptation logics on the foundation of dynamic SPL feature models. Thus, this thesis combines a dynamic SPL feature model for defining the possible reconfigurations with a self-adaptive system. This feature model should be integrated into the adaptation logic, and it facilitates the adaptation logic to plan reconfigurations based on context information of the managed resource. Finally, this should result in a method specifying the system features and reconfiguration behavior by an SPL engineer. This specification should be all the information needed for the adaptation logic to work.

The approach of this thesis is to develop an adaptation logic prototype incorporating the possibility to use dynamic SPL feature models. This prototype is assessed in a qualitative evaluation setup. This work uses a distributed computing use case for evaluation purposes as part of a simulation. The simulator mimics the complete distributed computing system. Additionally, the adaptation logic is tested using arbitrary feature models unconnected with the use case which tests it independently of the managed resource. For faster and more standardized development, the adaptation logic is implemented using FESAS which is a framework for developing self-adaptive systems [KVB13].

1.2. Structure

The remainder of this thesis is structured as follows: Chapter 2 introduces the fundamentals of self-adaptive systems and (dynamic) software product lines. The subsequent chapter shows related work in the domain of dynamic software product lines. The fourth chapter describes the actual adaptation logic approach for self-adaptive systems using (context) feature models. The following chapter presents a use case implementation of the adaptation logic employing FESAS, a framework for building self-adaptive systems. Chapter 6 outlines the use case evaluation. As already mentioned a distributed computation system namely the Tasklet system presented in [ESK+17] is the use case. The finishing chapter summarizes the findings of this thesis followed by possibilities for further research.

2. Self-Adaptive Systems and (Dynamic) Software Product Lines

This chapter introduces the fundamental concepts used in this thesis. The first section shows the basics of self-adaptive systems and adaptation logics. Then software product line techniques including the specification of variability are presented. The last section extends the static software product line methods with dynamism, leading into dynamic software product lines (DSPLs).

2.1. Self-Adaptive Systems

This section gives an overview about self-adaptive systems (or SAS) in general with a focus on the adaptation logic (AL). The following introduction of this section presents possible definitions and descriptions of the term self-adaptive system followed by an example.
Oreizy et al. give a definition for self-adaptive systems [OGT+99, p. 55]:

> *Self-adaptive software modifies its own behavior in response to changes in its operating environment. By operating environment, we mean anything observable by the software system, such as end-user input, external hardware devices and sensors, or program.*

Another definition is given by Laddaga et al. [LRS03, p. 1]:

> *Self-adaptive software evaluates its own behavior and changes behavior when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible.*

In comparison the definition of Laddaga emphasizes the "self-" and evaluation aspect of a self-adaptive system. The "self-" in self-adaptive means that the software system decides on its own (or autonomously, see [BDMSG+09]) to adapt its

behavior corresponding to a perceived change of the environment. This definition also mentions the idea to have an internal evaluation that tries to improve the system's performance constantly. The improvement of the performance is, e.g., possible using a learning-based component [KC03]. This may be achieved by the component finding correlations between system states and adaptations

Considering the terminology according to Salehie and Tahvildari many researchers use the terms self-adaptive system, autonomic system, and self-managing system synonymously [ST09]. Another perspective which this thesis uses is that self-adaptive systems are a subset of autonomic systems [ST09].

An example for a self-adaptive system could be an autonomous car that has to react to a traffic jam in front of it. This situation could result in replanning the route to the destination. A prerequisite is the system's ability to sense and understand its environment. Based on the information obtained, it has to plan and execute an appropriate action to react to changes. As seen in the example, the architecture of self-adaptive systems must be specialized for the purpose of moving decisions the system possibly needs to make towards runtime [BDMSG+09]. In order to achieve this shift several properties and components of a self-adaptive system have to be designed in a certain way which is described in the next section.

2.1.1. Structure of Self-Adaptive Systems

This section presents the typical properties and components of a self-adaptive system. The foundation property for self-adaptive systems is self-management [KC03]. Self-managing software results in a system that should work all the time without interruptions. This aspect frees system administrators from low-level tasks. As part of self-management there are four so called self-* properties: *self-configuration*, *self-optimization*, *self-healing*, and s*elf-protection* [KC03]. A self-configuration system intends to configure itself according to high-level policies of the overall IT environment. Thus, it embeds seamlessly into the IT environment. Self-optimization describes a learning component of the system. This component adjusts the adaptations for better results. There are two possible ways of adaptation: *parameter adaptation* and *compositional adaptation*. Parameter adaptation changes the system parameters while compositional adaptation changes structure, architecture, or both. Therefore, the self-optimization property of the system

constantly tries to change its parameters or composition over time to achieve the best possible results. This means the system is able to improve its performance on its own. Of course, problems can occur in this process. If problems arise, the self-healing mechanism comes into place. This mechanism tries to locate, analyze, and correct problems. The last component is self-protection. It should automatically detect and defend against attacks or cascading problems that could not be solved by the self-healing process. Additionally, it reacts to early reports based on sensor data to reduce the impact of arising problems. All self-adaptive systems are supposed to have these properties in common [KC03]. Still, their general structure can vary. There are two strictly different compositional approaches to build a self-adaptive system.

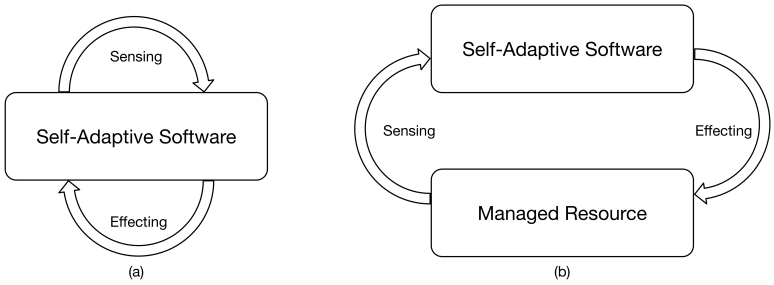

Figure 2.1.: Internal (a) and External (b) Adaptation Logic [ST09]

As seen in Figure 2.1 the structure of a self-adaptive system can be classified into *internal* and *external* adaptation logic [ST09]. This classification specifies how the actual system and the adaptation logic are combined. Either the adaptation logic is part of the main system (see Figure 2.1(a)), or it is designed as an external component (see Figure 2.1(b)) communicating with it. The internal approach is faster to implement and may be an option in very small systems. The maintainability is higher in the second approach. However, this approach needs communication between the AL and the managed resource. As the second approach is more scalable, exchangeable, and reusable, this is the broadly used method to implement self-adaptive systems [ST09]. Scalability is achieved, e.g., by having dedicated machines only for the adaptation logic. The independence of the external approach also makes it easy to use the same adaptation logic for multiple managed resources or to compare different adaptation logic approaches by exchanging them. This is not easily possible with an internal adaptation logic as the main system is composed together with the adaptation logic. Salehie and

Tahvildari have published a survey on self-adaptive systems in which no system uses the internal approach [ST09]. Thus, most self-adaptive systems consist of a separated adaptation logic and managed resource [BDG+13]. The adaptation logic constantly tries to adapt the system according to the information received from the managed resource while the managed resource provides the actual main functionality of the system. The managed resource can be a hardware or software component. The adaptation logic and the managed resource are connected in two ways. The adaptation logic sends adaptations to change the managed resource while the resource sends data about itself to the manager. This data can be sensorial or statistical. The adaptation is accomplished by either changing parameter values or by exchanging components [BDG+13, MSKC04]. As already mentioned the first possibility is called parameter adaptation, the latter one compositional adaptation. A system can be used as managed resource if it is able to provide sensor information and receive adaptation actions.

Finally, the important component of a self-adaptive system is the adaptation logic. The adaptation logic must sense changes, understand them, plan adaptation actions, and execute them. Thus, much research has been done to find effective ways to design this component. In the last years a common way in developing the adaptation logic has emerged which is presented in the next section.

2.1.2. Adaptation Logic

According to Brun et al. the generic way to achieve self-adaptation is to use feedback loops [BDMSG+09]. A feedback loop consists of four components: collect, analyze, decide, and act. This model is an advancement of the sense-plan-act approach taken from the early development of artificial intelligence. The collection component collects relevant data from the environment. The data could consist, e.g., of sensorial data or user input. With the data the adaptation logic should be able to determine the state of the system. The next step is to analyze the selected raw data. The analyze component structures the data and reasons about it using, e.g., models or policies. Based on this structured data the decision component determines how the system state may be improved. In this step it may be possible to use probability theory to conclude the best adaptation according to the current state. The act component then executes the adaptation by sending it

to the managed resource. Then the managed resource changes according to the received actions.

Kephart and Chess have used the generic control loop to develop an adaptation logic with four functional parts: *Monitor*, *Analyze*, *Plan*, and *Execute* [KC03]. The initial letters are the reason to call this approach the MAPE cycle. The MAPE cycle is embedded in a component called autonomic manager that represents the adaptation logic. The basic MAPE cycle starts with monitoring the raw data coming from the managed elements. Not only this mechanism fetches and monitors the data, but according to the MAPE approach of Brun et al. it also filters the data [BDMSG+09]. The analyze phase processes this prepared raw data. Metrics that violate constraints and the reasons for these violations are identified. The following planning phase determines necessary changes in order to get the best possible result for the system or to resolve any problem identified in the phase before. The execute part then simply executes the developed plan. These components communicate only via direct communication channels. Thus, in the plain MAPE approach there is no shared knowledge. Hence, no global history of states, events, and adaptations can be preserved. For this purpose the advancement MAPE-K has been developed. MAPE-K uses the same four components in the adaptation logic as MAPE [KC03]. The only addition is a shared knowledge base connected to the four components. This knowledge component can be used in the analyze and planning phase to compare current events with a history of events to find big changes in the state of the system. The differences found can be saved in the knowledge base for future reference [BDMSG+09].

Although the MAPE-K approach is a good guideline for developing self-adaptive systems, there is still no general approach for every environment and need. According to Brun et al. there is also a lack of a general possibility to model a system [BDMSG+09]. Additionally, it may be beneficial to use a middleware to generalize all parts of the system and make them more reusable. The concept of reusability is the most emphasized intention of SPLs. The next chapter introduces SPLs as well as their dynamic extensions: dynamic SPLs. The idea of software reusability in software product lines can therefore be related to the concept of self-adaptive systems with external adaptation logics.

2.2. Software Product Lines

This section presents the foundation of the modeling approach used for specifying the reconfiguration space of the software product. Also, it can be used to model the context internally and externally of the managed resource. In this section software product lines (SPLs) and their static configuration approach are presented. Section 2.2.3 specifically presents the feature diagram methods for modeling the feature models in a graphical way. Section 2.3 shows an extension for supporting dynamic feature selection at runtime: Dynamic software product lines (DSPLs). Based on the idea of dynamic reconfiguration in DSPLs the model type used in this work's approach is presented as well: Context-aware feature models. This advanced feature model type builds on top of the static feature modeling approaches presented in the following section.

2.2.1. Introduction to Software Product Lines

According to [HHSS08] the idea of software product lines emerged from general economics. Starting with the development of the conveyor belt by Ford the concept of *economies of scale* arose. Economies of scale "arise in the production of multiple implementations of a single design" leading to cost reductions [GS03, p. 17]. This mass production was cheaper, but did not have many diversification possibilities between the products compared to handcrafted individual items [PBV05, p. 4]. Based on this mass production the idea of reusing major parts of similar products that are only distinct in smaller individual parts developed. This approach is called Product Line Engineering (PLE) and the goal is economies of scope. Economies of scope means "efficiencies wrought by variety, not volume" [GJ83, p. 142]. The result of applying PLE are mass-produced but individualized products emerging in mass-customization. Davis defines this idea of mass-customization as follows: "Mass customisation is the large-scale production of goods tailored to individual customers' needs." [Dav87]. PLE facilitates companies in building up a generic platform that can be used as basis for all product variants. Reusability is the key here for the resulting cost reductions. The software development community became aware this idea emerging in the SPL method [HHSS08]. The tradeoff between handcrafted individual items and

mass produced products can be seen in software engineering as the difference between individual development and standard software [PBV05, p. 4].

The Software Engineering Institute of the Carnegie Mellon University defines SPLs on their website as following [SEI]:

> *A software product line (SPL) is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.*

The definition shows the original PLE idea of having a common platform and developing multiple individual features on top for meeting the needs of one specific area. This common platform is created using the so-called core assets. This shows an important step in SPL development: defining commonalities of the whole product line. This is one of the two SPL lifecycles which the next section introduces.

2.2.2. SPL Lifecycles

The two SPL lifecycles are domain engineering and application engineering. Both lifecycles require to already have business planning, product, and requirement information present. Then it is analyzed which common features apply to the whole product line and which features should be product specific. In the following the two lifecycles are introduced briefly. Figure 2.2 shows the whole SPL process.

The main goals of the *domain engineering* process are to define the commonality and the variability of the product line [PBV05, p. 21]. Commonality and variability are defined using a variability model. This defines common and exchangeable system parts. Additionally, the set of applications of the software product line should be defined. Each step should create reusable artifacts that employ the defined variability. These domain-specific artifacts compose the platform the software products rely on. The artifacts are connected by traceability links to retain consistency. This avoids inconsistent artifacts which may result in unusable or broken application products.

The *domain engineering* cycle begins with the *Domain analysis*. This includes requirements engineering to define and document the "common and variable re-

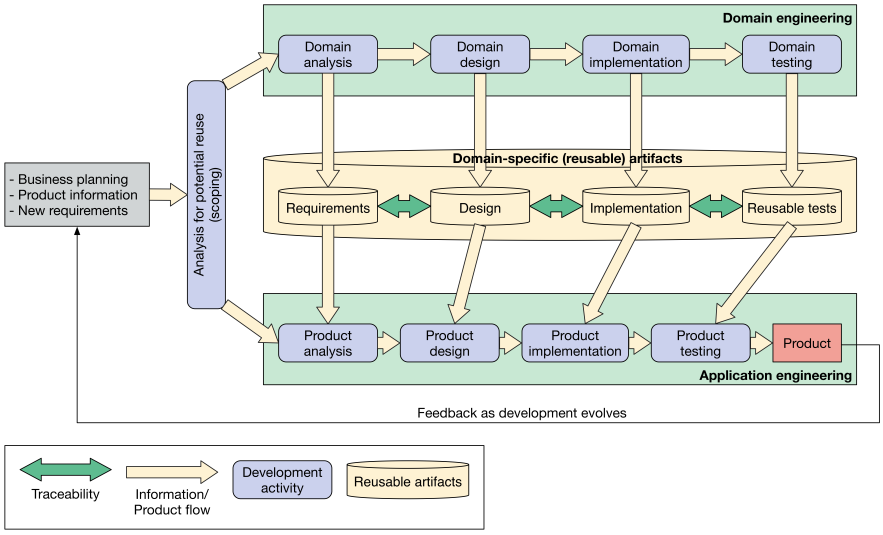

Figure 2.2.: SPL Lifecycles [HHSS08]

quirements of the product line" [PBV05, p. 25]. The most interesting product of the domain analysis process for this thesis is also created here: the variability model. It will be introduced in detail in the following section. *Domain design* should result in a high-level reference architecture usable for the whole product line. The requirements from the first step are the input for this step. Then the *Domain implementation* step should create concrete designs and implementations that are common to the whole SPL based on the reference architecture. *Domain testing* is a verification and validation step, checking all the steps that happened before. Furthermore, this measure should test the common artifacts to reduce errors in the common platform right from the start [PBV05, p. 27].

Application engineering aims to exploit the common platform of the SPL as good as possible and to relate the software product to the reusable domain-specific artifacts [PBV05, p. 21]. Additionally, it binds the variability model to the actual product instance that is to be built. *Product analysis* is also concerned with requirements engineering. Here the focus should be on identifying the differences between platform and product requirements. *Product design* uses the reference architecture to instantiate an actual product architecture and configuring it to the needs of the product. *Product implementation* should create the application as a combination of the common platform implementation artifacts and product specific modules. This results in the finished application exploiting as many

domain-specific artifacts as possible. The last step *product testing* runs tests on the finished software product. The outcome is a report with the test results. This ends the application engineering and results in the completely finished product. As seen in Figure 2.2 the products are used as feedback for possible new business planning requirements.

After this brief introduction of the whole SPL process the next section focuses on the models to define variability in the product line. These models are used later in combination with the planning in self-adaptive systems to define the reconfiguration behavior of a managed resource.

2.2.3. Variability Models

According to Pohl et al. variability models can be created using standard UML modeling techniques [PBV05, p. 75 f.]. However, since UML is not specifically designed for facilitating SPL development processes, so-called feature models are the common way in specifying features of a SPL. A feature is a "system property that is relevant to some stakeholder and is used to capture commonalities or discriminate between systems" [CHE04, p. 267].

Features are organized in feature diagrams. They are a tree structure representing the software system as a whole. The tree consists of a root feature with several layers of child features. A feature model generally consists of a feature diagram and additional information such as information on the binding time or priorities. Benavides et al. identified three major categories in the domain of feature models: basic feature models, cardinality-based feature models, and extended feature models [BSRC10]. In the following they are briefly introduced.

Basic feature models: Based on the literature review of Chen et al. most basic feature modeling approaches are based on the Feature-Oriented Domain Analysis (FODA) approach by Kang et al. [CAA09, KCH$^+$90]. Kang et al. were the first who introduced the term feature model and proposed a hierarchical feature tree structure for specifying all features of a SPL [BSRC10]. The original FODA notation includes the elements shown in Figure 2.3 (a). Also, features could require each other or could be declared as mutually exclusive. These properties are called cross-tree constraints. However, these properties were not depicted graphically yet. In the graphical representation simple text at the ends of the edges was

used for the features themselves. Later Kang et al. extended their original approach, e.g., by representing features as text boxes [KKL+98, SHTB06]. Parent features that have multiple child features are provided by either one or multiple of these child features. In this case child features specialize a parent feature. Furthermore, new elements were introduced to the original FODA notation later [GFD98]. They added an *or* operator as well as graphical representations for the cross-tree constraints. These new elements are depicted in Figure 2.3 (b).

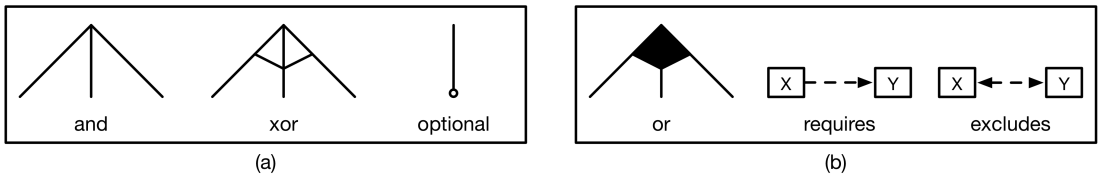

Figure 2.3.: Basic Feature Diagram Elements: Original FODA Notation (a) [KCH+90] and extended FODA Notation (b) [GFD98]

Cardinality-based feature models: Riebisch et al. propose that implicitly there are UML-like multiplicities covered by feature models [RBSP02]. In order to improve the understanding and to formally define them they introduce an annotation for representing the multiplicities of feature sets. Later these cardinalities were defined more concretely as group type cardinalities [CHE04, BSRC10]. A group type cardinality further defines the case when a parent feature is part of the system, how many child features are allowed in a configuration. As an example a group type cardinality of *0..** means that the child features are all optional. Also, there are feature instance cardinalities which denote how many instances of a feature can exist at runtime [CHE04]. For distinguishing both cardinality types, group type cardinalities are denoted with angle brackets and feature instance cardinalities use square brackets (see Figure 2.4 (a)). Analogously to the UML notation a cardinality is annotated with a lower and an upper limit. In summary, cardinalities state in a very clear way how to interpret a feature diagram. They enhance the overall expressivity and possibilities to exactly state the needed constraints.

Extended feature models which, according to Benavides are also called advanced or attributed feature models, are able to express additional attributes on features [BSRC10]. There is no consensus on the information an attribute should contain. However, most approaches state that an attribute should contain a name,

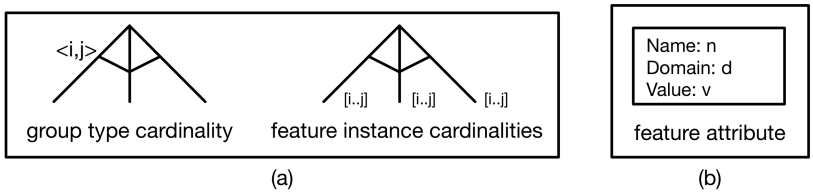

Figure 2.4.: Extended Feature Diagram Elements: Cardinality-based Notation [RBSP02, CHE04] (a) and Feature Attributes (b) [BTRC05]

a domain, and a value. Using these attributes it is possible to, e.g., describe requirements for a certain feature more concretely. Since there are multiple approaches for describing attributes it is also not clear how to depict them. This thesis uses the notation introduced by Benavides [BTRC05]. The notation can be seen in Figure 2.4 (b). Attributes can also be related to other attributes and can express conditions for certain features to be only available as part of a configuration, if an attribute has a certain value.

This section presented the generic and static SPL approach as well as extensions of the FODA notation for feature diagrams. Based on this introduction Dynamic SPLs as well as context-feature models are introduced in the following section.

2.3. Dynamic Software Product Lines

Due to the demand of today's environments, adaptability gets gradually more important for software systems [HHSS08]. Static SPLs do not fulfill this requirement as the variation points defined in feature models get bound at design time. The difference between SPL and DSPL binding can be seen in Figure 2.5 (a) and (b). A software product built using the SPL approach is configured once. The first step is to apply the feature model for selecting overall valid configurations from all possible configurations. Then a configuration for the product to be built is selected. Thus, the developer builds such a variant for a static execution environment. In this case the software runs fine in a rather static environment. In the case of a dynamic context, SPL based software possibly does not perform well anymore due to the requirement for adaptation and reconfiguration at runtime [CBT+14]. Software build within a DSPL is able to adapt itself, e.g., to changing user preferences or contexts. This is realized by binding variation points at the start of the software and at runtime repeatedly. You can see this in Figure 2.5

(b). Like in the SPL approach the feature model is applied for selecting valid configurations. Then a valid start configuration is selected at design time. In the DSPL approach the valid configurations are connected by arrows building a directed graph. The product changes its configuration based on a graph defining possible transitions between all valid configurations. This enables adaptive behavior for the software. As configuration changes are triggered by the context, it is crucial to monitor the context while always storing a model of the current system and the state of its environment. In fact, in order to plan a good reconfiguration it is the most important task for the application to monitor itself and the context and change the configuration based on the monitoring results [HHSS08].

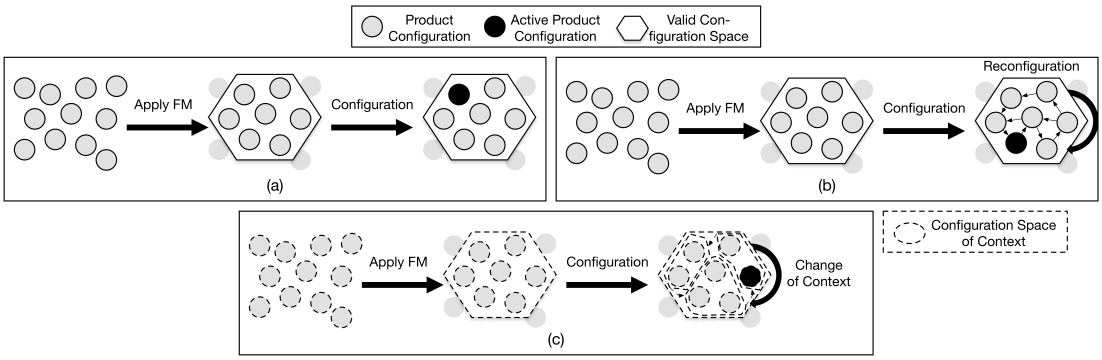

Figure 2.5.: SPL (a), DSPL (b) and Context-aware (c) Configuration [SLR13]

The context monitoring and modeling can be divided into the closed and open (world) approach [AAL10, BHA12]. At the same time, Abbas et al. coined the term Autonomic SPL. The name emerged from the fact that it features a MAPE-K loop (see 2.1.2). The closed approach means that the possible states of the DSPL get fully defined at design time. This can be done either by hand or by offline training. The context states are stored together with the best configurations in a (dispatch) table. In the open approach the system is supposed to find new context situations and configurations at runtime. According to [AAL10] and [BHA12] this is usually tackled with an online learning approach with an own MAPE-K loop on top of the first MAPE-K loop. This can be seen as adaptation of the adaptation logic or as self-improvement [KRP16].

The next chapter presents related work in the domain of DSPL approaches. For better classification a taxonomy is introduced first. Then an overview on DSPL approaches is given.

3. Related Work in Dynamic Software Product Line Approaches

[BBD16] provide a rather comprehensive overview over DSPL approaches. They categorize them into multiple dimensions of two taxonomies: an adaptation and a DSPL taxonomy. In the following the dimensions of both taxonomies are presented briefly. Then a brief introduction on all DSPL approaches of the overview is given. The taxonomies are used later to categorize the approach of this work while the other DSPL approaches are used to show the state of the art in this field.

3.1. Adaptation and DSPL Taxonomy

The first section presents the adaptation taxonomy [BBD16]. For better understanding you can find the optional Figure A.1 in the appendix. It shows the whole adaptation taxonomy in a tree structure. The adaptation dimension is divided into the sub dimensions *goal*, *cause*, and *mechanism*.

3.1.1. Adaptation Taxonomy

Goal: The goal is divided into the *goal type* and the *goal evolution*. Goal type refers to the four adaptation goals by [KC03]. It states the aim of the adaptation runtime. [BBD16] omitted self-protecting here because of the "unavailability of a DSPL engineering approach with this goal type" [BBD16, p. 7]. Thus, the goal type can be either *self-configuration*, *self-optimization*, or *self-healing* (see Chapter 2). The goal evolution can either be *static* or *non-static*. Static means the system has a fixed adaptation policy and fixed number of variant while non-static systems can learn completely new policies and goals at runtime.

Cause: The cause sub dimension is only characterized by a *cause type*. The cause type can be the context, the system, or the user. This determines what

can be the cause for an adaptation. While in the context and the system some observed variables may change, the user is, e.g., capable of providing new goals at runtime. One example for a system cause can be a breakdown of a component.

Mechanism: The mechanism is divided into *mechanism autonomy* and *mechanism type*. The mechanisms determine what kind of changes an adaptation logic is able to make. The mechanism autonomy can either be manual or autonomous. This specifies if an adaptation has to be triggered by an external party or if the system decides itself that an adaptation is executed. The mechanism type can be *code*, *component*, or *architectural*. Code means to, e.g., set new parameters on the code level. The architecture stays exactly the same. This is a very limited approach as adaptation logic and managed resource are strongly coupled. The component approach substitutes complete components with the same interfaces. This makes it easy to build a plugin-based system. The architectural method changes connections between components, or it introduces new ones. Also, new components could be included as well.

3.1.2. DSPL Taxonomy

The following DSPL taxonomy is structured using the standard MAPE-cycle [BBD16]. Thus, the first dimension distinguishes between *monitoring & analysis*, *planning*, and *execution*. Again, for a better overview a tree showing all aspects presented here is depicted in Figure A.2 as part of the appendix. Since the focus of this work is the planning aspect of SAS the corresponding description is more detailed compared to the other dimensions.

Monitoring & Analysis: This category is divided into *context model*, *context reasoning model*, and *context sensing*. The context model can either be a simple *property-set* which represents the context using preselected properties that are used for planning or an *ontology* using, e.g., the Web Ontology Language (OWL) [W3C04]. OWL can be used to model the context using high-level structures that also state relationships between context elements. The context reasoning model is divided into *rule-based logic* and *query languages*. Rule-based logic uses, e.g., propositional logic for context reasoning. It is a simple method but with too many rules this approach is not efficient anymore [BBD16]. Query languages can be used to reason on context data more efficiently as the scope for reasoning

(e.g., by adding a time frame which is relevant to the user) can be specified very precisely [PAG06]. Especially when ontology techniques such as OWL are used query languages like SPARQL can be helpful for complex context reasoning [W3C08, BBD16]. Context sensing is split into *observation* and *notification*. The first method requires the planner to fetch data, e.g., from a context manager while the latter makes sure the context manager informs the planner when a change in the context happens.

Planning: Planning has the dimensions *variability space model*, *planning model*, *planning level*, *planning type*, and *transformation*. The variability space model can be an *enumeration*, *variation points*, or *feature models*. When the enumeration approach is used all possible variants are enumerated in the first place. This method is generally used for simple systems with a limited amount of variants. Variation points are specified by selecting single points in the base system where components can easily be swapped. This approach is also very simple. However, it does not allow constraints between variation points. The feature modeling approach is the most powerful approach which is used to specify all features of the system including variation points and constraints. Feature models can also be augmented to context variability models. In addition, they link context states to the system features using constraints to limit the feature model configuration space. The planning model can be described using *state transition diagrams*, *event-condition-action (ECA) rules*, and *utility-functions*. According to the authors [BBD16] state transition diagrams can only be used together with the enumeration variation space model. The states are variants and the transitions possible adaptations. ECA rules trigger an action if certain conditions hold true. Events are changes in the context, conditions are certain thresholds, and actions are activations and deactivations of features. The problem concerning this approach is that with a high number of rules there can easily be conflicts which are hard to notice. The most complex but also most advanced approach is the use of a utility function. This function calculates the desirability of a system state using context information. The variant with the highest expected utility is always chosen. The biggest challenge here is to define such a utility function that covers all needed aspects. The planning level can either be on the basis of *features* or on the whole *architecture*. Planning on the basis of features separates the context requirements from the actual realization while the architectural approach

mixes both concerns by specifying concrete architectural changes. The planning type can be either *rule-based*, *goal-based*, or *utility-based*. Rule-based is usually modeled using ECA rules or a state transition diagram. Thus, the actions are clearly specified. Goal-based planning is implemented using high level goals that the planner has to decompose to sub goals and finally distinct actions. As it is not easy to satisfy multiple goals at the same time the desirability of a state can be described more easily using a utility. Hence, in the utility-based approach the action with the highest expected utility is chosen as adaptation action. Transformation, the last dimension, is categorized into *direct link*, *aspect model weaving*, and *transformation rules*. Direct link means there is an exact mapping of the feature model to the architectural model. Thus, changes in the feature model directly influence the architecture of the system. If this mapping does not exist, direct link is not possible. Aspect model weaving is a model-driven development method that allows creating detailed architectures from high level models. Using this method selected features are woven into the base system. The Transformation rules method needs an additional architecture feature model that models all variants of the architecture model. The mapping between the two feature models is represented by transformation rules. The rules can, e.g., be modeled in propositional logic [BBD16].

Execution: Execution is categorized using the dimensions *architecture model*, *architectural style*, *variation entity*, and *runtime reconfiguration*. The architecture model is the abstract view of the system representing all variants. This model is used to identify parts in the system that should be changed by an adaptation. The first representation approach could be *custom languages*. As the model should always be up-to-date at runtime it is efficient to use a domain-specific language. Another possibility is to use business *process modeling languages* like the business process modeling language together with the business process execution language. The last possibility is to use *architecture description languages* which model the architecture of a system on a very high level. Usually this exists at design time so it can be reused by the planner. Architectural style is divided into *component-based*, *service-oriented*, and *service component architecture (SCA)*. Component-based means there are fixed connections between components, and the components can be exchanged. Additionally, using specific patterns runtime variability of components themselves can be implemented.

Service-oriented architectures are based on the same idea as SPLs that mostly everything should be reusable. The coupling is very loose in this approach. Changes can be implemented by turning services on or off or by changing communication channels between existing services. The last approach is the SCA which is a hybrid approach. Here functionality is provided by components with interfaces as if they were services. The components interact by calling services methods on other components. This combines the strengths of the first two approaches. Variation entities define the actual parts of the system that get changed when an adaptation is executed. Changes can be carried out on the basis of *components*, *services*, *aspects*, or *connectors*. Components can get deployed independently. One problem here is that a single component often does not simply represent one feature. Thus, granularity is a concern. Another challenge is to maintain the system states when components get replaced. Services have the same problems as components but are more loosely coupled. It is easily possible to start new services and use them, since they are defined by interfaces. The aspects dimension relates to aspect-oriented programming. Here a mapping between features and aspects of the system is needed. These mappings can get easily unmanageable. The last possibility for variation entities are the connectors that can be changed between other entities. These connectors can be "glue codes, communication channels, or workflows" [BBD16, 24]. The dimension runtime reconfiguration or middleware defines how the system is reconfigured and how the configuration state is represented. The authors propose that mostly an application-independent middleware should take care of the adaptation itself and it should represent the configuration state. This can either be implemented using a *component model* or using *dynamic aspect weaving*. A component model defines the semantic and the syntax of components as well as their composition. Examples for such a system are OpenCOM or OSGi [CBG+08, BCL+06]. Dynamic aspect weaving means to change the current aspect of the system. This can be used when the variation entities are aspects.

After this presentation of the DSPL taxonomy by Bashari et al. [BBD16], the following chapter shows the DSPL approaches that get categorized in their work using the adaptation and the DSPL taxonomies.

3.2. Overview on DSPL Approaches

In this section each approach included in [BBD16] is briefly introduced. Then their findings in terms of the characterization of the approaches based on their adaptation taxonomy are presented. You can find the results presented here additionally in Table A.1. As the DSPL taxonomy is very detailed, the categorization of the approaches concerning this taxonomy is omitted here. You can find the results concerning this taxonomy in Table A.2 and in [BBD16].

Service-Oriented Dynamic Software Product Lines [BGL+12]: They use the Common Variability Language (CVL) in combination with the Business Process Execution Language (BPEL) and aspect-oriented programming [Inc06, KLM+97]. The so-called 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 DyBPEL is used. DyBPEL augments the ActiveBPEL execution engine with additional aspect-oriented variability possibilities. As part of this DyBPEL engine there is a coordination component. The coordinator gets the change request from the variability designer. It triggers a BPEL modifier for changing the execution inside the embedded ActiveBPEL engine as well as a runtime modifier migrating running processes. The authors use a smart home use case. However, since they propose a BPEL based approach any business process using the Business Process Modeling Notation (BPMN) could be used. Obviously and according to the literature review of [DDD+13] this approach mainly addresses the execution of the MAPE-cycle. According to the taxonomy presented in the previous section Bashari et al. state that there is no goal type. The evolution is static, and the cause is clearly the user as everything is user-triggered [BBD16]. As the user starts adaptations the mechanism autonomy dimension is manual, and since aspect-oriented programming is used, which does not change components or the structure, the mechanism type is code-level.

Genie: Supporting the Model Driven Development of Reflective, Component-based Adaptive Systems [BGF+08]: The authors of this approach developed a tool called *Genie* which supports the development and modeling of reconfigurable systems that are component-based. 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 recon-

figuration behavior. The environment states are regarded as state diagram with transitions between the states. In [BGF+08] and [BSBG08] they, e.g., propose a use case in the domain of embedded sensor networks as part of a flood warning system. There they use a model with two boolean context variables and one system variable for determining in which of three states a flood detection system should work in. According to [BBD16] the goal type is self-configuration. The presented use case shows this in a clear way as the sensor network changes, e.g., its power management based on information on possible flooding. For example, when heavy rainfalls occur the system communicates more often and using more power to detect a possible flooding situation faster. Still, the evolution is static as no new configurations are learnt at runtime. The context is the reason for an adaptation. Also, the adaptation mechanism acts autonomously. As a feature model is used it works component-based. This results in exchanging different components at runtime.

Applying Software Product Lines to Build Autonomic Pervasive Systems [CFP08]: Cetina et al. developed an approach in the domain of pervasive computing or more specifically in the domain of smart homes [CFP08]. They use a model driven development methodology for modeling the features and the behavior. For the features they are using the feature model in the notation of [BTRC05]. The structure of the system as well as the configuration behavior is described using the PervML modeling language which is specifically designed for the application in the domain of pervasive computing. 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. On the basis of the smart home use case they specifically addressed three scenarios: A resource becomes available, a resource becomes unavailable, and a user has a new goal. [BBD16] state that the goal type is self-healing due to the fallback mechanisms as well as self-configuring. It employs static goal evolution. The cause for an adaptation is always the context. The adaptations are autonomous and component-based since the system decides on its own to adapt and it uses a feature model-based approach.

MADAM (mobility and adaptation-enabling middleware) [FHS+06]: The system is represented as component model where components conforming to a matching interface can easily be 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 the components with the highest expected utility are implemented. As the approach uses a utility function for evaluating its behavior the goal type is characterized as self-optimizing. The evolution is static. The adaptation cause can be either the system or the context since both are represented in the middleware. It works autonomously and in a component-based way as well.

REPFLC (Reconfigurable Evolutionary Product Family Lifecycle) [GH04]: The presented lifecycle is divided into three parts: Product family engineering, target system configuration, and target system reconfiguration. Product family engineering is similar to the static SPL approach. This step includes product family architecture creation consisting of components and connections as well as the specification of variations of the product family. So-called reconfiguration patterns for the runtime adaptation are created as well. These specify possible transitions between configurations. It is divided into state and scenario model. The state model describes the changes when an adaptation happens and the scenario model defines certain conditions when an adaptation should be triggered. Target system configuration is also comparable to the SPL approach as the system gets deployed on a target system based on certain requirements. The target reconfiguration step finally uses the models specified at design time to support runtime adaptation of the product. Concerning the taxonomy only three dimensions could be specified. The goal evolution is static, and it is autonomous and component-based.

DiVA (Dynamic Variability in complex, Adaptive system) [MBJ+09]: DiVA is based on four metamodels that get exchanged inside the system. The four models are a DSPL model, a context model, a reasoning model, and 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, e.g., with ECA rules. These rules trigger adaptations. The architecture model can be any architecture model such as a standard UML model or an SCA model. These models are used in a three layer architecture. The bottom layer contains the application logic while

the top layer contains the adaptation logic. The middle layer connects the layers providing context data to the top layer and for configuring the bottom layer. A reasoner picks the best configuration based on the context information. A consistency checker for the models used at runtime concludes the approach. In fact, this allows a non-static goal evolution. It also works autonomously and in the component-based way.

Context awareness for dynamic service-oriented product lines [PBD09]: Parra et al. use the context-sensing middleware COSMOS which provides so-called context nodes. A context node has always the same interface and provides the context information of one sensor each. A context model is the basis for storing the current contexts. Each so-called context-aware asset is, e.g., defined by some value that is to 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. It is self-configuring and static. The cause is always the system and it is autonomous as well as component-based.

With the exception of one approach, all methods work component-based and autonomously. The same applies to the goal evolution that is mainly static. The majority acts on changes in the context while having the self-configuration goal type. Most approaches require that the DSPL developer has to learn a new modeling technique which is not used in other contexts. Also, some approaches are focussed on special use cases at the moment which may make them not easily applicable in other domains. Thus, this thesis uses the method of Saller et al. [SLR13] as foundation for a use case independent approach for modeling the possible configurations of a managed resource. They propose the idea of incorporating an additional context model into feature models. Thus, they use a notation that is familiar to every SPL aware software engineer. Also, it is possible to use a simple mapping between context and features using standard cross-tree constraints. Hence, there is no need to learn any new modeling language or mapping between context and features. The following section introduces the idea of context-aware feature models in more detail.

3.3. Context-Aware Feature Modeling Approach

As already mentioned Saller et al. take the generic feature models as a starting point [SLR13]. Additionally, to the actual feature tree in this approach the context is modeled in a similar way as second child node from the system root node. The context features are created the same way as the features at design time. Thus, the model is called context feature model (CFM). Saller et al. use feature and group cardinalities as well as feature attributes in this approach (see 2.2.3). The reconfiguration behavior is constructed using cross-tree constraints. These constraints are created between a feature attribute of a context variable representing one state of the variable and a feature from the feature sub-tree. At runtime the context feature attributes get assigned by real context information. This should happen using some context-aware engine [SLR13]. The process of DSPL reconfiguration is changed accordingly as described in Section 2.3 using Figure 2.5. Additionally to the application of the feature model, the context determines possible configuration spaces to which possible reconfigurations are assigned. As one context can allow multiple configurations the dotted lines in the figure show possible configurations of one context situation. The arrows between the contexts indicate transitions between the different context situations. These transitions happen with a certain transition probability which could be taken into account for the reconfiguration decision. This could enable proactive behavior. It is assumed that each context represents a distinct state of the environment. It is possible that multiple configurations are possible at the same time in each context situation. This adds the possibility of conflicts between multiple reconfiguration decisions.

This chapter presented related work in the field of dynamic software product lines. The two taxonomies according to [BBD16] have been explained. The seven approaches the authors compared according to these taxonomies were shortly introduced and categorized using the adaptation taxonomy. Finally, the context feature modeling approach by [SLR13] is introduced which is the foundation of the adaptation logic developed in this thesis. The approach that was developed as part of this thesis is presented in the following chapter.

4. Approach of Planning Reconfigurations using Feature Models

This chapter is an introduction to the approach of this thesis for planning reconfigurations of an SAS using DSPL context feature model methods presented in Chapter 2. First satisfiability problems which are the foundation for generating plans with certain constraints are briefly introduced. Then an overview about the approach is presented.

4.1. Satisfiability Problems

Petke defines (boolean) satisfiability problems as following [Pet15, p. 15]:

> *The problem of deciding whether there is a variable assignment that satisfies a propositional formula is called the Boolean satisfiability problem (SAT).*

The logical knowledge of a SAT problem consists of multiple clauses constructing the so-called conjunctive normal form of a formula [Ben04] [RN09, p. 253 ff.]. Each clause consists of one or multiple literals while a literal is a "propositional variable or its negation" [Ben04, p. 124]. Thus, each clause represents some constraints. All clauses which are disjunctions for themselves are conjunctively connected with each other. If there is a possible assignment of boolean values to all literals satisfying all sentences there exists a so-called model for this setting. According to Russell and Norvig satisfiability of a sentence is defined as follows [RN09, p. 250]:

> *A sentence is satisfiable if true in, or satisfied by, some model.*

This boolean satisfiability problem can be solved by SAT solvers. A SAT solver gets clauses as input and creates a model for this input if there is any [RN09, p.

271 f.]. The mostly used input and output format for SAT solvers is the DIMACS CNF format [Sat09].

Another problem type are constraint satisfaction problems (CSPs). In fact, SAT problems are a subset of CSPs [Pet15, p. 1]. Thus, CSPs are not limited to boolean satisfiability, but they can solve problems with arbitrary values assigned to the variables [RN09, p. 202 ff.]. CSP solvers can find solutions to a given set of variables, constraints on them, and partial values on the variables. It is possible to map a SAT problem to a CSP problem and the other way around [Wal00, Ben04, LBL08]. As an example the naive approach for converting a CSP to a SAT problem is to create a boolean variable for each concrete value and add mutual exclusivity to one group of variables representing a range of values.

The third existing problem type are satisfiability modulo theories problems (SMT problems). SMT solvers which can handle this kind of problems are solvers including background theories for interpreting more complex problems [BSST09]. This makes it possible to solve so-called first-order logic problems including quantifiers and arithmetic operations [BSST09]. Since this high level of expressivity is not needed for finding solutions to the feature model problems stated in this thesis, SMT solvers are not explored any further at this point.

Since CSP problems can get converted naively to SAT problems reducing the complexity of the solver, a SAT solver is used in the approach of this thesis. However, in future work it may be possible to also include a CSP solver and some heuristics for selecting one of the solvers. Right now only a SAT solver is used. In the following section the big picture of the approach is presented.

4.2. CFM-based MAPE-K Cycle

As a starting point the idea is to augment a MAPE-K cycle based adaptation logic with a CFM inside the knowledge component. Furthermore, rules for relating raw sensor data to context feature attributes as well as mappings relating features and feature attributes to their literal representations for the SAT solver are part of the knowledge. The SAT mapping is built directly at the start of the system. This facilitates the knowledge component to return the corresponding literal given a feature or feature attribute. Additionally, so-called priorities and

costs reside inside the knowledge component. They are used for conflict resolution, and the selection of one configuration given multiple configurations are possible. Priorities and costs are present for all system features which are part of a feature group. A priority is a number stating the priority of a system feature inside its corresponding feature group. 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. In the following the complete data flow through the adaptation logic is briefly illustrated.

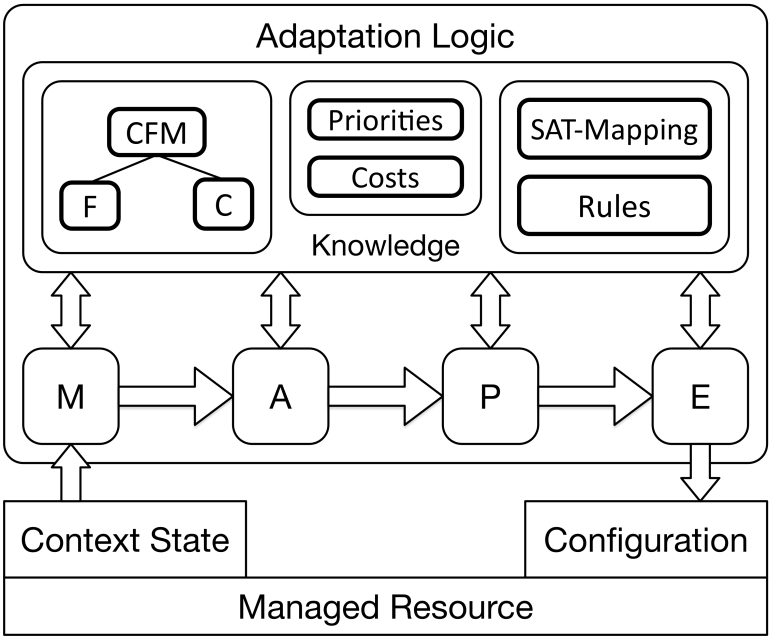

Figure 4.1.: MAPE-K Cycle using a DSPL Context Feature Model and additional Information

Figure 4.1 shows an overview of the architecture of this thesis' approach. 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 state information. This data can consist of internal or external context information related to the managed resource. The adaptation logic may receive raw sensor data which means it may need to be formatted, filtered, or aggregated inside the adaptation logic. The monitoring component receives the data, prepares it and passes it to the analyzer component. The preparation here means to interpret some serialized input like an XML or JSON string (for XML and JSON see [BPSM+98, Cro06]) in order to create plain data objects for working with

the sensor data. Also, it is possible to receive raw text input from the managed resource which has to be parsed in the first place. The resulting data objects can be used more easily by the analyzer component than a raw string.

The monitoring information gets analyzed by the analyzer component. This adaptation logic is able to receive and process partial sensor data until the managed resource sends a message indicating the data of one tick or run is sent. Thus, the monitor gets sensor information not as one package but as single sensor information. The analyzer must create the average of all single entries in order to create one single sensor value representing the average system state. This average system value is used to map the values to actual context feature attributes representing the context state of the system. At first 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 averaged values to actual context feature attributes. The resulting attributes which are selected according to the context information are forwarded to the planner component. The approach supports partial knowledge meaning not all context feature attributes must be present. Hence, even without full knowledge the system is capable of finding a configuration.

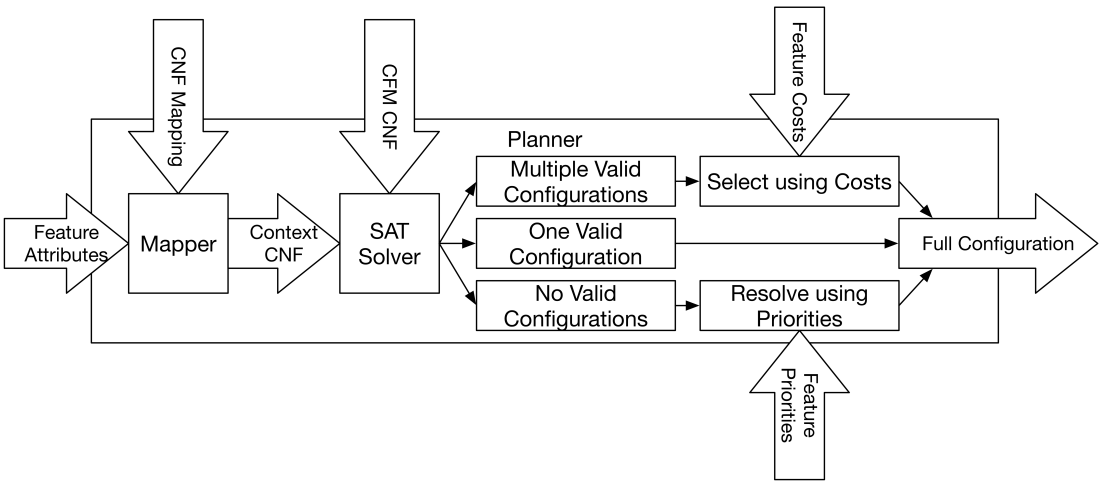

Figure 4.2.: Internal workflow of the planner component

Figure 4.2 shows the complete workflow of the planner component. The planner is the most important component for this approach as it contains the whole

planning logic with a CFM as foundation. As the planner mainly works with a SAT solver the result of the first step should be a logical representation of the context information in conjunctive normal form (CNF). In fact, DIMACS CNF is used here (see Section 4.1, [Sat09]). This first step works using the SAT or CNF mappings as part 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 conjunctive normal form. The result is the 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. Figure 4.2 shows this internal process of the planner component. If there is only one valid configuration, the planner is finished as there are no configuration options it can choose from. In case of multiple possible configurations the additional costs 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. If a conflict happens there may be two or more system features in conflict. In this case the priorities 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 home automation system. One context feature requires to open the windows when it is hot in the room. Another context feature requires the system to activate the sprinklers when there is fire in the room. In this case the sprinklers should obviously have a higher priority. The result of the planner is a complete list of system features the managed resource should activate. The selected configuration is sent to the execute component which makes sure that the change in the configuration gets deployed properly. This ends one complete cycle through the adaptation logic. The next chapter shows implementation details of this approach.

5. Implementation

The previous chapter briefly introduced the method for planning reconfigurations based on DSPL feature models used in this thesis. As with any MAPE-K cycle based approach, the monitor gets raw context sensor data and the analyzer analyzes the data. This analyzing step requires mappings in order to successfully map the input data to context feature attributes in the CFM. The planner mainly features a SAT solver and works based on the information that is available inside the knowledge component. All information that is needed for planning is encapsulated inside the knowledge component. If the CFM is conflict free, only costs have to be added to the CFM in order to evaluate different possible configurations when multiple configurations are valid. If there is the possibility for conflicts, priorities for features are also needed. This approach uses a CFM model with additional information to plan on the basis of a satisfiability solver. This means that no special modeling technique or implementation is needed to use this approach. In the end, the planner produces one single configuration that is deployed to the managed resource by the execution element.

This chapter introduces implementation details of the MAPE-K cycle approach presented in the previous chapter. The first section demonstrates the FESAS framework (framework for engineering SAS) that is used in this work for the implementation. The second section describes the MAPE-K components of this thesis' approach implemented as FESAS components.

5.1. Introduction of FESAS

The aim of the FESAS project is to establish a generic and reusable framework for developing self-adaptive systems. Today's self-adaptive systems are mostly tailored to their environments and needs [KVB13]. Thus, the framework should consist of reusable components and processes that can be utilized for all different kinds of problems and system domains. The framework is model-driven. This

means a system designer is able to set up a design model and the framework translates this design model into an adaptation logic that can be executed by the FESAS runtime. Therefore, a system designer can use high-level tools instead of starting low-level from scratch. This innovative approach could speed up the needed time to develop a complete self-adaptive system. The system itself is constructed using predefined generic components in a reference architecture. This architecture uses the MAPE-K cycle (see Section 2.1.2). The designer can replace the adaptation logic elements with elements from the same type. For this purpose FESAS has a adaptation logic element repository the designer can choose components from. This enables a system designer to build custom adaptation logic cycles using the elements in the repository.
Each part of the MAPE-K cycle is implemented as a separate component. This supports the idea of Kephart and Chess stating that autonomic systems are a collection of autonomic elements to deliver services to users and other autonomic elements [KC03]. As these components should be able to be distributed over several different devices FESAS also supports an object-oriented middleware in its reference architecture to achieve stable interoperability in distributed systems. Still, it is possible to use FESAS only on one machine without any middleware in between. This means standard method calls would suffice in this special case. At the beginning FESAS used the BASE middleware which was developed by Becker et al. [BSGR03]. BASE is implemented in Java. This ensures the possibility to use it on every Java capable device. BASE allows devices to propagate their available local services to other devices via peer-to-peer techniques. If a device needs to consume a certain kind of a service it searches in the propagated service registry to find a running instance on a remote device. BASE is very customizable and expandable with plugins. The customizability makes it possible to strip down BASE to the very needed components to fit it even on devices that only run the Java Micro Edition. Hence, FESAS can run on smallest devices when BASE is used. However, FESAS is not limited to this particular object-oriented middleware [KRVB15]. It is part of one reference architecture.
FESAS makes it possible to use either parameter adaptation or component adaptation [KVB13]. This provides flexibility in the concrete adaptation possibilities. Additionally, it provides marshalling and unmarshalling capabilities for the data that is exchanged between the adaptation logic elements. This also improves the

component exchangeability possibilities. FESAS uses JSON as serialization notation in the reference system (for JSON see [Cro06]). This JSON data gets passed from one component to the next. Each MAPE cycle component consists of meta information about the component and the actual implementation in Java. The meta information consists of simple elements like a name and a description as well as supported and produced information types. In essence, components that support the same information type should be able to be exchanged without any problems. The actual implementation of most of the components mainly consists of a `callLogic` method. The JSON data is passed to this method encapsulated inside a high-level `KnowledgeRecord` object. The logic of the component does whatever it is intended to do and passes its result to the next component in the cycle by calling a generic `sendData` method. This method also takes care of the marshalling of the data. Then the next component in the cycle is called.

The simulation of this work is embedded in the context of the FESAS project and the proposed reference system using the BASE middle. FESAS increases development speed and it is completely use case independent. Hence, FESAS is a helpful development tool for SAS development. The MAPE cycle elements of the simulation are implemented as FESAS components. The following section presents details of the generic CFM-based adaptation logic implementation inside FESAS.

5.2. Implementation using FESAS

As the FESAS framework is implemented in Java the implementation of the approach also uses Java. The implementation of the prototype system has been conducted using the FESAS IDE [KRB+16]. The FESAS IDE is an add-on for the popular Eclipse IDE. In this thesis only the FESAS Development Tool which is part of the FESAS IDE is used. The FESAS Design Tool for the orchestration of different MAPE components and for specifying the decentralization of them is not used in this work. The reason for this is that the MAPE-K components in this thesis are fully centralized. There is no decentralization present. The FESAS Development Tool enables the developer to easily create skeletons of adaptation logic components as well as the meta information that is needed for publishing components to the FESAS repository.

For creating the MAPE component, the FESAS IDE is used to create the initial skeletons. These skeletons consist of a Java implementation file and a meta information JSON file that is used for the repository. The Java skeleton consists of some auto-generated meta information as well as an `initializeLogic` and a `callLogic` method. The most important method of an adaptation logic component is the `callLogic` method that is called by the FESAS runtime environment at execution time. FESAS triggers the `callLogic` method of each component providing the input data which is the previous component's result. This can be triggered using the `sendData` or `sendArrayList` methods which let FESAS provide the data to the next component in the chain.

In the development phase, a test mode can be used. Using another wizard of the FESAS IDE it is possible to create these logic tests. For each test input data in the JSON format can be specified. This makes it easy to debug the system. For running the system using FESAS, the MAPE-K components can be exported into compressed zip files for deployment inside a FESAS repository.

For illustration purposes an entire run through the adaptation logic is described using the data center use case presented in [KRVB15]. It describes self-managing data centers that start servers given a high work load. Accordingly it should stop server given a low workload. The monitoring only observes the workload of the servers. The configuration determines if a start policy, a stop policy, or a keep policy should be activated. In this example one adaptation logic manages three data centers. Figure 5.1 shows the big picture of the data flow through the MAPE components. The monitor component receives sensor information about the workload from each of the three data centers 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, introduced in Section 5.2.1) which is a Java object. Based on this input the planner selects the start policy. The result is sent as a string to the executor which forwards it without modification. In the following, implementation details with reference to the example on all MAPE-K components are explained.

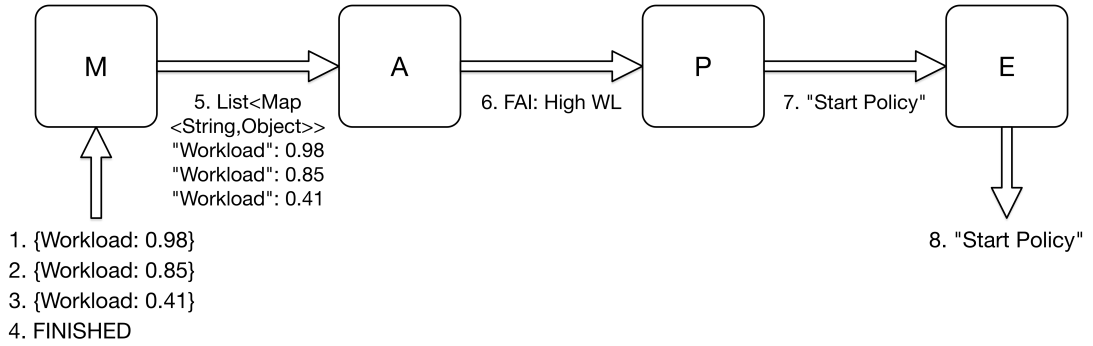

Figure 5.1.: Complete Data Flow between the MAPE Components. FAI means `FeatureAttributeItem`

5.2.1. Knowledge

The first component that is presented in the following is the knowledge component. Figure 5.2 shows the UML diagram of the knowledge metamodel. It contains all aspects of knowledge that are needed for the adaptation logic. The SAT mapping mentioned in the previous chapter is not shown in this diagram as it is generated at the start of the system dynamically.

The diagram shows that the actual context feature model is only one part of the knowledge. The knowledge class also has so-called feature attribute rules, the priorities, and the costs. Starting with the feature model class it has the reference to all abstract features of the model. An abstract feature may have a parent feature and (multiple) children. As the feature type is abstract a feature has to be either of the type `ContextFeature` or `SystemFeature`. These feature type classes exist to better understand and structure the model. Thus, they provide no distinct functionality. Additionally, features can have two types of cardinalities: a feature instance cardinality and a group type cardinality (for cardinalities in general see Chapter 2). The feature instance cardinality specifies how many instances of a feature are allowed to be present in the system at runtime. The group type cardinality states the type for the feature group of the feature's children. Features implement the `IConstraintElement` interface that marks classes which can be used to express constraints. Thus, constraints consist of two `IConstraintElement` objects. The `Constraint` class is also abstract. At the moment only require constraints are implemented. Thus, this is the only

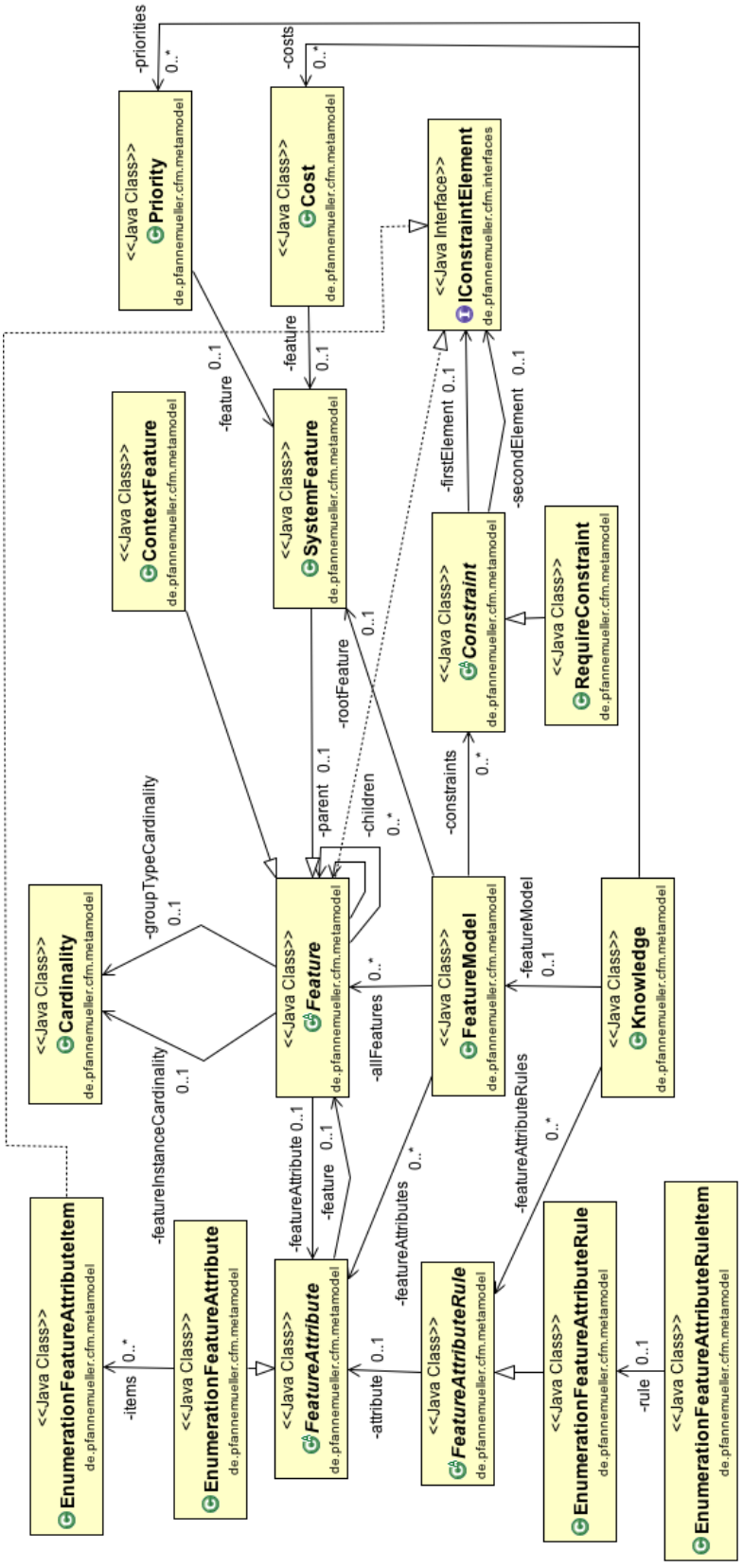

Figure 5.2.: UML diagram of the knowledge metamodel

subclass of the Constraint class. However, it is easily possible to add the exclude constraint if there is the requirement in the future. In this case, another subclass of the class `Constraint` is needed. This subclass simply has to be handled when the CNF of the CFM is created. According to the extended feature model approach, features can also have attributes. Thus, the abstract feature has the possibility to add such an element to it. Again `FeatureAttribute` is an abstract class. At the moment the system only supports enumerations as attribute type meaning, e.g., integer or double values are mapped to multiple attribute items representing value ranges. These attribute items also implement the `IConstraintElement` as they are supposed to require certain system features. There is the possibility to, e.g., add floating point or integer feature attributes later if needed. 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 has to exist. A rule specifies the name for the sensor input matching it to an attribute. Additionally, it provides a matching method for determining the attribute item that is represented by some context value.

For importing the knowledge of the specified system to the adaptation logic it is possible to use a serialized input file. At the moment the system is able to import files conforming to the JSON representation of the knowledge class. However, the file format is a secondary matter. The `IKnowledgeParser` interface specifies how a file parser has to work. Thus, implementing this interface adds the possibility to parse other arbitrary file formats such as XML easily. Facilitating the idea of having no fixed data exchange format the data transfer object (DTO) pattern is realized in this implementation [Fow02, p. 401 f.]. The DTO pattern provides an abstraction between the actual Java objects representing the knowledge and the file format of the input file. The DTO representation of the actual Java object is simplified and handles issues such as circular references. This happens if the original Java representation is passed to a serializer without any modifications. Thus, for every Java object from Figure 5.2, a DTO object exists as well.

In FESAS itself a so-called `KnowledgeManager` class is used for storing knowledge data. It implements the singleton pattern and stores the `Knowledge` class as well as the SAT mapping created when the system starts. It is initialized with the path to a knowledge file.

5.2.2. Monitoring

The monitoring component gets multiple raw strings. The component is build to convert the raw JSON data into a hash map with strings and objects representing the raw data. This simplifies the following processing steps. These entries are added to an array list until a stop command is received. Referring to Figure 5.1 the stop command is "FINISHED". This approach makes sure that all information of one tick or run of the managed resource is sent. The three JSON strings are collected in a `List<Map<String,Object>>` object. After the stop command this array list is passed to the analyzing component waiting for new sensor data. This triggers the analyzer and the monitor waits for new sensor data.

5.2.3. Analyzing

The analyzing component gets the array list serialized as JSON from FESAS. It gets transformed back into an array list. Then the average of all values and entries is calculated. Referring to the example in Figure 5.1 the analyzer creates the average of the three `Map` objects. 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 *High WL* is selected. Then the FAIs are sent to the planning component as array list.

5.2.4. Planning

This component is the core contribution of this work. Thus, the section of the planning component is most exhaustive. Section 4.2 described the general planning approach (see Figure 4.2) without specific information on the implementation. The actual Java implementation uses Sat4J as SAT solver [LP10]. Code Listing 5.1 shows the callLogic method of the planning component. Lines 5 and 7 parse the JSON text to the original array list containing context features attributes and maps them using the SAT mapping to their literal representation. Otherwise the SAT solver is not able to work with them. Line 9 checks if there is a model using the given context information. At this point the solver already has loaded the CNF of the context feature model. If there is no

model, the conflicting features are determined in line 10. The conflicting features are passed to the so-called `PriorityConflictSolver` which implements the generic `IConflictSolver` interface. This facilitates the idea of different conflict solving components that can be exchanged. In this thesis, only the `PriorityConflictSolver` is present. The `PriorityConflictSolver` iterates through all constraints of the conflicting context features to find the actual conflicting system features. For each feature group with conflicts the priorities from the knowledge component are used. Thus, the feature with the highest priority of every conflicting group is selected. Adding the resolved feature selection as well as the remaining conflict free context information to the solver ends the conflict resolution (line 16). The lines 18 through 20 are called in case of no conflict. In this situation the context information is simply added to the solver. Lines 22 through 28 are concerned with getting all possible models based on the context feature attribute selection. Every model is added to one array list. In order to get all possible models after each model that is found by the solver, the negation of this model is added to the clauses list of the solver. For the case with multiple possible models, a class implementing the `IConfigurationSelector` interface is needed to decide which configuration should be activated. In this approaches' implementation, a `CostConfigurationSelector` uses the costs assigned to the system features as part of the knowledge. For every feature group with a decision, the feature with the lowest cost is selected. The following lines remove the unnecessary context features and inner features from the list. If there is only one configuration possible the resulting configuration is selected. A string array list containing all system features that should be activated is sent to the executing component in the end (line 51).

```
1  @Override
2  public String callLogic(IKnowledgeRecord data) {
3          ⋮
4          this.resetSolver();
5          String arrayListData = data.getData().toString();
6          ⋮
7          int[] contextSatLine = this.getContextSatLine(arrayListData);
8          // If there is no model, resolve the conflicts first
9          if(!this.solver.isSatisfiable(contextSatLine)){
10                 int[] conflict = this.solver.unsatExplanation(contextSatLine);
```

```
11              // Use PriorityConflictSolver
12              IConflictSolver conflictSolver = new PriorityConflictSolver();
13              ArrayList<ArrayList<Integer>> list =
14                      conflictSolver.getResolvedContextSatLine(
15                      contextSatLine, conflict, KnowledgeManager.getInstance());
16              this.solver.addClauses(list);
17      }else{
18              for(int i = 0; i<contextSatLine.length; i++){
19                      this.solver.addClause(contextSatLine[i]);
20              }
21      }
22      ArrayList<int[]> allPossibleModels = this.solver.getAllPossibleModels();
23      ArrayList<ArrayList<ISatElement>> featureLists = new ArrayList<>();
24      for(int[] model: allPossibleModels){
25              ArrayList<ISatElement> featureList =
26                      this.mapper.getSatElementListFromSatSolution(model);
27              featureLists.add(featureList);
28      }
29      ArrayList<String> featureNames = null;
30      if(allPossibleModels.size() > 1){ // Check if multiple models are possible
31              // Use CostConfigurationSelector
32              IConfigurationSelector configurationSelector =
33                      new CostConfigurationSelector();
34              ArrayList<SystemFeature> featureList =
35                      configurationSelector.selectConfigurationFromFeatureList(
36                              featureLists, KnowledgeManager.getInstance());
37              // Clean list
38              featureList = this.removeInnerFeatures(featureList);
39              featureNames = this.getNamesOfFeatureList(featureList);
40      }:
41      else{
42              // Select first configuration
43              ArrayList<ISatElement> satElementList = featureLists.get(0);
44              // Clean list
45              ArrayList<SystemFeature> featureList =
46                      this.filterSystemFeatures(satElementList);
47              featureList = this.removeInnerFeatures(featureList);
```

```
48                  featureNames = this.getNamesOfFeatureList(featureList);
49          }
50          // Send list with active system feature names
51          this.sendArrayList(featureNames);
52          return "Planner sent feature selection to executor";
53  :
54  }
```

Listing 5.1: `callLogic` Method of the Planner Component

5.2.5. Execution

The execution in this case only forwards the feature selection to the managed resource. Referring to Figure 5.1 the feature selection is a string with the system feature names separated by a comma. The managed resource must map the system feature names to actual policies and actions. The executor performs no additional actions.

6. Evaluation and Discussion

This chapter is concerned with the evaluation of the CFM-based adaptation logic approach in the simulation of a use case. At first the use case is introduced. Then information on the use case implementation is presented. Afterwards, the setup for the actual evaluation is given. This includes three evaluation questions that should be answered. The evaluation results are also presented in this section. The last section discusses findings of the results as well as properties of the system according to the DSPL guidelines from [BBD16].

6.1. Use Case Description

The use case for testing the CFM-based adaptation logic approach presented in the last chapter is concerned with the so-called Tasklet system [ESK+17]. The idea of the Tasklet system is to provide a middleware-based infrastructure for distributed computing on heterogeneous devices [ESK+17]. The Tasklet system consists of three entities: resource providers, resource consumers, and resource brokers. Providing resources to the Tasklet system means to offer so-called Tasklet virtual machines (TVM) that are able to run byte code which is based on C–. Resource consumers send their byte code for computation to providers for remote execution. Additionally, it is possible to be provider and consumer at the same time, meaning there is not necessarily a remote execution [ESK+17]. Brokers provide the hybrid peer-to-peer infrastructure for this endeavor. Brokers are specialized infrastructure nodes in the system, and form a peer-to-peer overlay network themselves. Each resource provider registers at a broker. Bootstrapping is used here to initially find the entry into the overlay network. Consumers do not register at brokers. They send requests for providers to any broker they know. An overview of an example overlay network topology is shown in Figure 6.1. It shows the inner broker overlay network, the connection between providers (P), consumers (C), and brokers as well as the execution of Tasklets between two

pairs of nodes. Tasklets are sent directly from a consumer to a provider. This underlines the hybrid peer-to-peer approach. It also shows the possibility that one entity can be provider and consumer at the same time (P/C). Each entity in the network runs the Tasklet middleware [ESK+17].

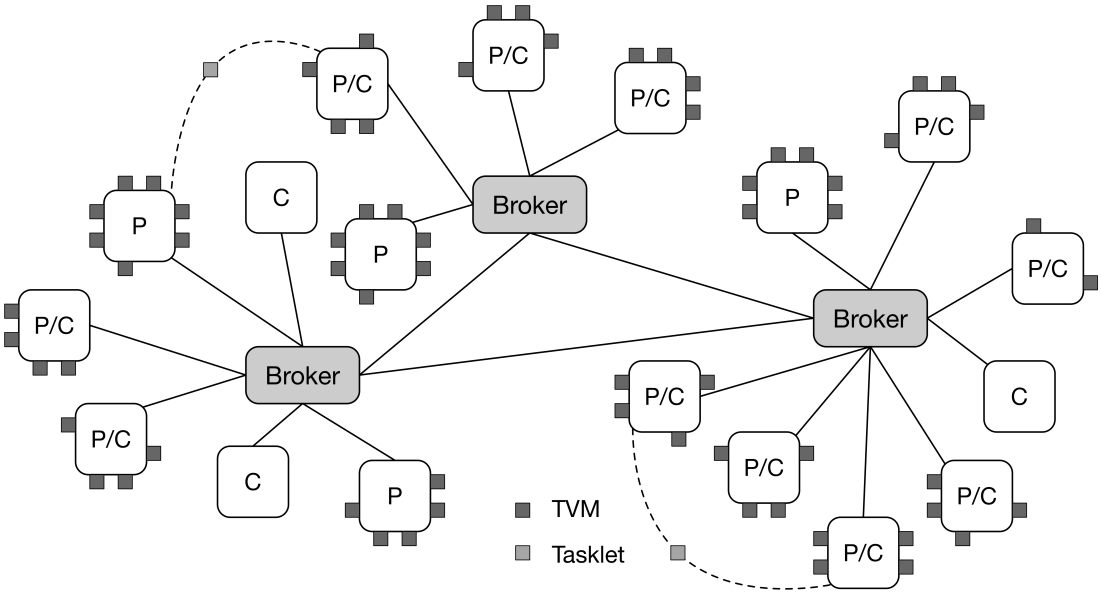

Figure 6.1.: Schema of a Tasklet Network Topology [ESK+17]

The middleware handles the construction of the C– code, the execution, and the distribution of Tasklets. Tasklets can be created with special flags. As one example a Tasklet can be flagged with a speed requirement which forces the middleware to execute it on a fast provider node accordingly. For more details on the approach refer to [ESK+17].

6.2. Context Feature Model Development

Based on the use case the evaluation of this work is developed. Since the broker network is not able to manage itself at the moment in terms of the number of brokers or the global distribution of the resource providers, the idea of this work is to create a broker management system. In order to find possible ways to optimize the system, 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. The model was created in discussions with the Tasklet system developers Janick Edinger and Dominik Schäfer as well as in cooperation with Markus Weckesser from the Technical University of Darmstadt who is a researcher in the field of dynamic software product lines and context feature models. The result of this process is the context feature model shown in Figure 6.2. In the following, the whole model is explained in detail. The subsequent sections introduce context and system features of the figure.

6.2.1. Context Features

At first the context features are presented to understand the purpose of the system and the system features more easily. They are shown on the right in the CFM figure (Figure 6.2). All context features are the average values of all brokers. Each context feature has a context feature attribute. Here each attribute uses the enumeration type.

The first attribute is the *fluctuation* in the whole system. It measures how many providers in a certain time span enter and leave the system in percent of all connected entities.

The next attribute is the *provider latency*. It quantifies the latency of providers to their corresponding brokers in milliseconds. This is important as it affects the overall execution time of Tasklets. A high latency could mean that a resource provider has a very long physical distance to the location of the broker. The distribution of brokers should be as good as possible in terms of latency (see the system features in the following).

The third attribute is the *percentage of high-performance requests*. Based on the complete number of Tasklet requests coming in at a broker, it represents the percentage of explicitly marked as high-performance requests.

The last context feature attribute is the *broker load*. This is an important metric as it determines if a a new broker is needed in the Tasklet overlay network. High load on the brokers may result in performance degradation concerning response time when a consumer requests a provider.

The value ranges for the attributes have been selected by performing multiple test runs of the evaluation system. The typical attribute values have been gathered

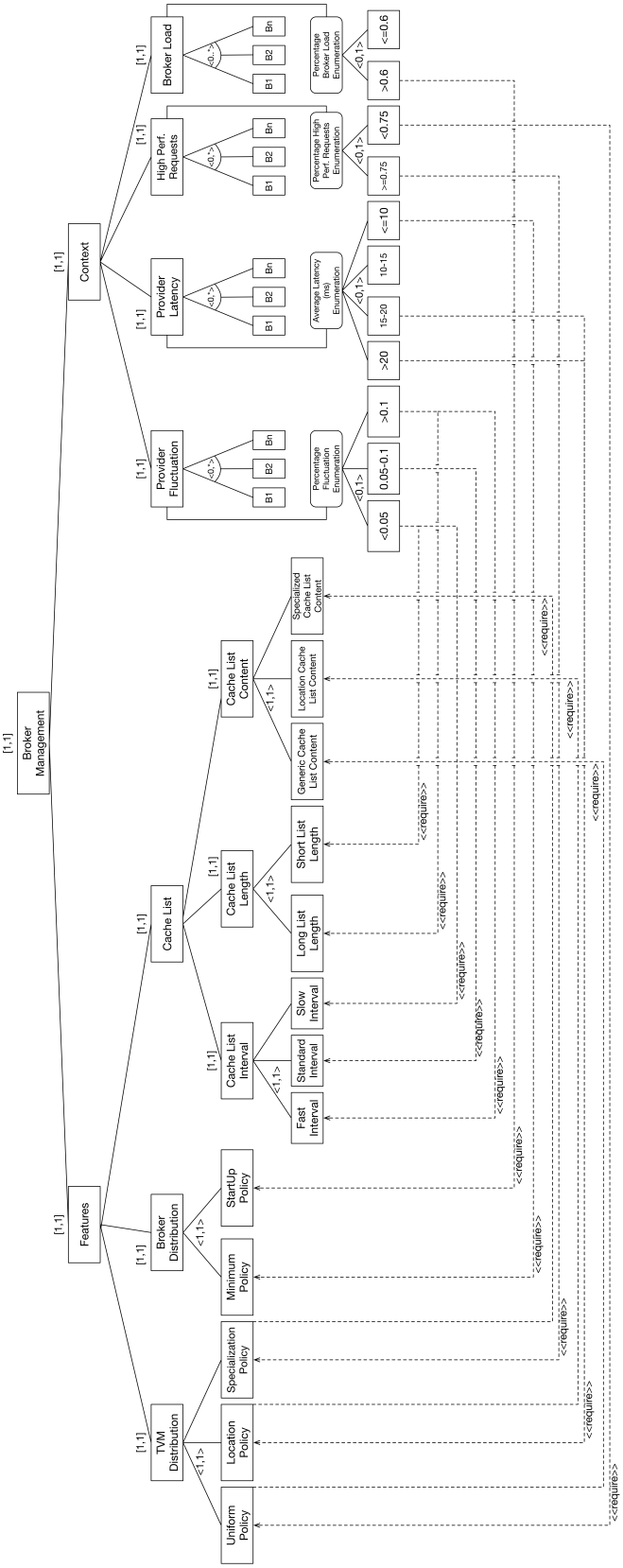

Figure 6.2.: Broker Management System Context Feature Model

for them. The ranges are selected for best presentiveness of the system.

Finally the diagram also shows that all context features have to be present at runtime. However, the context feature attributes may be not specified as the group type is set to $< 0, * >$. Thus, the model supports the idea of partial knowledge as stated in [SLR13].

6.2.2. System Features

Now the system features are presented. In order to minimize the number of provider requests to the brokers, cache lists are used. These lists are distributed at regular intervals to the consumers. Consumers poll for the lists all the time. They can be configured in terms of the distribution interval, the length meaning the number of providers on the lists, and the order of the lists' entries. The *Cache List Interval* can be slow, standard, or fast. When there is a high fluctuation in the network, the interval for the distribution of the cache lists are changed accordingly between these intervals. The *Cache List Length* can be short or long. Thus, when the fluctuation is high, the cache list length is increased to the long length, which means more providers are on the list. Hence, the consumer has more providers for establishing a possible connection before a request to a broker has to be sent. If the fluctuation is low, the short list length is activated accordingly. The *Cache List Content* is either generic, location-based, or specialized. Generic means the list is only ordered according to the reputation of the providers in decending order. It is triggered by the *Uniform Policy*. The location-based cache list puts providers that are in the same geographic region as the consumer at the beginning of the list. It is required by the *Location Policy*. The specialized cache list puts high-performance providers to the top of the list. This should result in a faster execution time when the percentage of high-performance TVM requests is high. The *Specialization Policy* triggers this.

The next system feature for the broker management system is the *TVM Distribution* feature. This feature is concerned with the distribution of TVMs across all brokers in the system. There are three possible policies forming an XOR feature group: Uniform Policy, Location Policy, and Specialization Policy. They are described in connection with the cache list content as they trigger their corresponding cache list content type. The Uniform Policy should uniformly dis-

tribute all TVMs. This policy should also take the speed characteristic of the TVMs into account. Referring to Figure 6.2 it is triggered when less than 75% of the provider requests are requesting a high-performance TVM. Additionally, it causes the Generic Cache List content that is introduced in short. The next TVM distribution policy is the Location Policy. If the latency is higher, the location policy is required. Since latency may imply the physical distance to the broker, the TVMs are moved to brokers that are in the same geographical region. Additionally, the cache lists are changed accordingly. Referring to Figure 6.2 the Specialization policy is required when more than or equal to 75% of the TVM requests need a high-performance TVM. This policy is supposed to move the high-performance TVMs to one broker that is specialized on high-performance requests. This should result in less communication between the brokers when TVM requests have to be forwarded when already all high-performance TVMs are busy at one broker.

While the TVMs can be distributed in the system somehow, the brokers can also be distributed and managed. This leads to the last system feature: *Broker Distribution*. For this feature there are two policies: the *Minimum Policy* and the *Start Up Policy*. The Minimum Policy tries to minimize the number of brokers. Hence, the broker with the lowest number of registered providers is stopped. The connected providers and consumers are moved to another broker with the lowest latency. Looking at Figure 6.2, if the overall latency in the system is lower than 10 ms, in average this policy is required. Also, according to the figure, the Start Up Policy is triggered if the overall average broker load exceeds 60%. All system feature groups are XOR groups since they are marked with the cardinality $< 1, 1 >$. All inner system features have to be instantiated at runtime according to the feature instance cardinality $[1, 1]$.

This section described the context feature model of the Tasklet system that is used to show the functionality of the CFM-based adaptation logic method of this work. The next section presents the actual implementation of the simulation system including a detailed introduction of every Tasklet component.

6.3. Simulator Evaluation

The use case is implemented as a simulated system. The initial implementation is carried out in [ESK+17]. Thus, this thesis extends the simulator with additional functionality. The largest additions to it are a new entity called broker manager as well as the implementation of the system features. Additionally, multiple new message types for communication between the different entities are added. The first subsection introduces the simulation in general. The subsequent subsection presents the structure and execution procedure of the simulation system. Then the third subsection describes each component of the simulation system in more detail. The last subsection shows the interface between the AL and the simulation and what additional elements were needed to make sensing and effecting possible.

6.3.1. Introduction

[ESK+17] present and use a simulation for evaluation purposes. They use it to evaluate the Tasklet scheduler with multiple different scheduling strategies. The simulator is capable of simulating high quantities of distributed Tasklet network participants. This simplifies the evaluation process and is cheaper than running an actual testbed. Additionally, measuring the performance of the scheduler is easier. The system includes the possibility to simulate failing nodes or network connections [ESK+17]. The simulated system of this thesis uses this state as foundation. The next section describes the implementation of the complete simulated system.

6.3.2. Implementation

At first the focus is on the simulator at its most important components. These components are a message queue and a list of all Tasklet machines. At the start of the simulated system the simulator creates a list of Tasklet machines. A configuration file specifies the quantities for each entity type. After this initial generation the simulator performs the first simulation step. It is a discrete simulated system. Hence, the performance of the system executing the simulation does not influence the results. Performing a simulation step includes three tasks for the simulator:

Sending the corresponding messages to every Tasklet machine, starting the execution of each machine's specific task, and collecting possible messages from all machines.

Each Tasklet machine has predefined steps it has to perform. At first each machine handles the incoming messages according to their types. Then it has to execute the `specificTasks` method that executes entity specific tasks. The machines can add messages directly to the message queue of the simulator.

So far, this section describes the simulator from [ESK+17] without the modifications which are part of this thesis. Figure 6.3 shows the described steps as well as the additional steps this work adds to the simulated system. The capability of sending status information about the connected resource providers from the brokers to the adaptation logic is added. The figure also shows the new broker manager entity that handles the selected configuration of the AL. When all machines including the brokers are finished with their tasks it sends the *FINISHED* keyword to the AL.

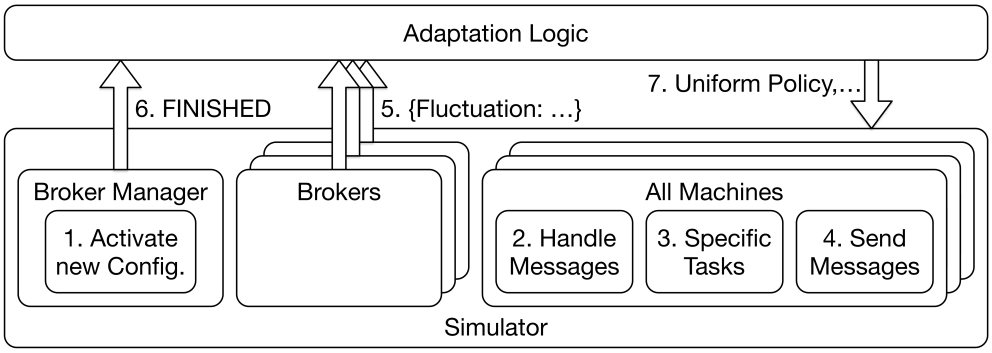

Figure 6.3.: Data Flow in Simulated System and AL based on [ESK+17]

This concludes the overview of the simulation system implementation. The following subsection introduces each Tasklet entity type in detail.

6.3.3. Simulated System Components

This subsection describes each Tasklet entity type. At first it presents the entities that are present in the original simulator of [ESK+17]. This includes resource consumers, providers, and brokers. Then it outlines the single broker manager entity.

Resource consumers have to handle *heartbeat messages* and *results* from providers, as well as *potential provider messages* and *cache lists* they get from brokers. The heartbeat messages are used to determine if a provider executing a Tasklet that was sent earlier is still active. It is used for monitoring the execution. When a result arrives, the consumer informs the broker. This way the brokers accumulate knowledge on reputation and reliability of the providers. Potential provider messages contain a provider corresponding to a provider request. Cache lists contain a list of providers. This list prevents the consumer from constantly sending provider requests to the broker. It can rather directly send Tasklets to providers of the list. The specific tasks of a resource consumer mainly consist of generating new Tasklets and sending them directly. If no cache list is present the classic way of sending a provider request to the broker is used.

Resource providers have to handle *Tasklet messages*, *provider status requests*, and *broker change messages*. For each Tasklet message it checks wether it requires a high-performance provider. If the provider does satisfy it, the Tasklet is executed and will finally be sent as a result to the consumer. Otherwise, this provider creates a new provider request for the consumer which created the Tasklet. This is sent to the broker of the provider. Provider status requests are used by the actual system feature implementations to fetch information about the providers in the network. This information includes if the provider is a high-performance provider as well as its geographical region. Broker change messages contain a new broker ID the provider should connect to. The provider specific tasks are to send heartbeats to its broker and consumers of whom it executes Tasklets for. Also, it returns results of finished Tasklets to the consumers.

Resource brokers handle heartbeat messages from providers, reports from consumers, so-called forwarding messages, provider requests, and providers transfers. The heartbeat messages from the providers are used for statistical and reputation knowledge about them. The reports are used for the same purpose. Forwarding is a special message type that is used when reports or provider requests are forwarded by another broker. This happens when a broker change takes place while a Tasklet is still executed. The resource broker receives provider transfers when a provider changes its broker and the statistical data should be retained. Brokers can get requests for sending their knowledge of a provider to another broker or get exactly this knowledge from another provider. Provider requests

are answered with potential providers. The specific tasks of a broker contain the task of managing the status and reputation of all connected providers. Additionally, cache lists are sent to consumers. The most important specific task is sending monitoring data to the adaptation logic. The resource broker passes it to the broker manager. The monitoring data is determined by every resource broker in every tick. It contains all context feature attributes that are part of the CFM presented earlier. The fluctuation is calculated by comparing the list of registered providers from the last step with the current step. The latency is created artificially. Each machine entity gets a random region at the start of the simulated system. Based on these regions the distance between the regions of the providers and the broker is calculated. There are three possible regions: AMER for America, APAC for Australia, eastern Asia and south-east Asia, and EMEA for Europe, Africa, and the Middle East. For example, if the region of the broker and a provider is the same, the latency is set to a value between 5 and 10 ms randomly. Other combinations have a higher latency ranges accordingly. The number of high-performance requests is counted in percentage of all requests while the load is directly represented by the number of active providers per maximum number of supported providers of the broker. The maximum number of supported providers is also set at the start of the simulated system.

Broker manager is the only entity type that has one single instance at runtime. The broker manager is a new entity in the Tasklet network. In this simulation there is only one single broker manager entity. Since this represents a single point of failure, a single broker manager would not suffice in a real world implementation of the Tasklet system. The brokers would run on separate servers providing the Tasklet infrastructure. This would be the same for the broker manager. This new entity contains instances of all policy classes and a list of registered brokers. Each new broker directly registers at the broker manager. The manager enables configurations it receives from the adaptation logic and it sends away the monitoring data of each broker. It only handles forwarding messages from brokers and the provider status messages. The former is divided into reports that need to be forwarded, and provider requests. Reports are forwarded by a broker if the provider which is part of the report is not connected with it anymore. Then the broker manager broadcasts such a report to all registered brokers. In contrast to this, provider requests are sent only to the subsequent broker in the list of regis-

tered brokers. As written above, provider status messages are used as knowledge resource for the actual system feature implementations. Thus, these messages get delivered to all active policies in the broker manager. The specific task of the broker manager contains the code for the execution of all active policies. In order to see the result of the policies for better judgement, the policies are only activated according to a fixed interval.

The next subsection presents the interfaces between the adaptation logic and the managed resource which is the broker manager in detail.

6.3.4. Connecting Simulator and Adaptation Logic

The simulated system is implemented in Java. Hence, it is easy to connect it to the adaptation logic which is also written in Java using FESAS. Figure 6.4 shows an overview of the simulated system including the AL and the interfaces between the two structures. As this Tasklet machine has a special role in the system, it is depicted separately from the other machines.

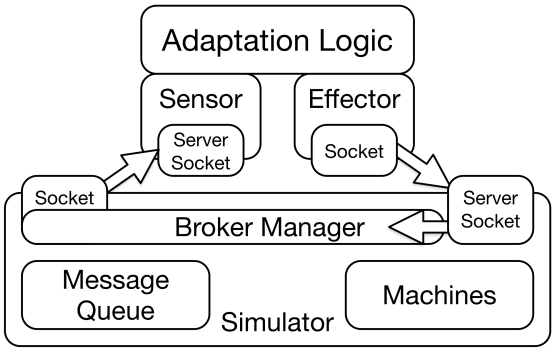

Figure 6.4.: Simulated System based on [ESK+17] with Interfaces to the AL

As already mentioned, the simulator is discrete. Thus, the adaptation logic is built into the simulation circle. This is done by executing the broker manager entity always at the end of one step after all other entities have finished their tasks. The data itself is sent using the `Socket` and `ServerSocket` classes which are part of Java. To achieve a use case independent implementation in the adaptation logic there are two additional FESAS components used: the sensor and the effector component. The sensor component starts a ServerSocket for listening for monitoring data on a specified port. The broker manager opens the counterpart

socket for sending the data. The sensor passes it to the monitoring component. Accordingly, the effector component gets the result of the executor and sends it by using a socket connection directly to the simulator. The simulator, which has an open socket as well, passes this configuration directly to the broker manager.

The adaptation logic consisting of the MAPE-K loop can remain the same throughout the process. There are no use case specific changes needed. The knowledge component just needs the path to the input file containing the complete knowledge definition of the system (as described in Section 5.2.1). The simulator is changed to wait for a new configuration before triggering the next simulation step. This assures the discrete execution. There are conventions on the data that is sent using the sockets. The first socket connection from the broker manager to the sensor has to send JSON data. The names for each JSON attribute must match the name of the context feature attribute it refers to. Also, the "FINISHED" keyword stating the end of one batch is fixed at the moment. The second socket connection sends a comma-separated list of strings which have to match the names of the implemented system features.

The discrete property of the simulated system facilitates the integration of the socket interfaces into the adaptation logic. The socket connections are a simple solution for connecting the adaptation logic and the simulation. After the presentation of the whole use case and its implementation, the next section introduces additional implementation results.

6.4. Additional Implementation Results

This section briefly presents other implementation results than the knowledge metamodel and the adaptation logic components. These are needed for conducting the evaluation.

KnowledgeFactory: To be able to test arbitrary context feature models, a so-called KnowledgeFactory has been built. It has three inputs: the number of knowledge objects to be created, the number of system features, and the number of context features. It creates as many enumeration feature attributes as context feature leaves are present. For each enumeration feature attribute two to five feature attribute items are created. Feature attribute rules are constructed with

ascending integer values. Constraints between feature attribute items and system features and between pairs of system feature are created randomly. Additionally, random costs and priorities are created. The resulting knowledge objects are converted to their DTO representation. This DTO representation is converted to JSON and finally saved to the disc.

ModelTester: The model tester can be used to test the knowledge data generated by the `KnowledgeFactory` using random context data. In fact, it tests the complete adaptation logic with generated knowledge data. Since the value range of the context feature attributes is fixed with the number of feature attribute items, random values fitting the value range are created for the monitor. It is used to measure the time from sending the random values to the monitor until a feature selection is generated.

6.5. Evaluation Setup and Results

This section elaborates the evaluation setup and its results. The first subsection presents the setup for evaluating the CFM-based planning approach using the Tasklet use case. This includes three evaluation questions that should be answered. The evaluation results are presented in a separate section. The succeeding section discusses the results.

6.5.1. Evaluation Setup

The evaluation is supposed to help answer the following three questions:

1. Is there any improvement using the adaptation logic for managing the broker management system?
2. Does the approach also work with a high number of participants in the network? How do the results differ compared to results with fewer network participants?
3. How much time is needed for a complete run of the adaptation logic with different randomized models with random context input?

Evaluation Question 1: For measuring possible improvements two run configurations in this first scenario are used. One configuration consists of the simu-

lation system connected to the adaptation logic. The other one is without the adaptation logic using a fixed configuration. 1000 Providers, 500 consumers, and two brokers are the start configuration. Each run ends after 10000 finished Tasklets. The number of needed ticks to finish them is one measurement. Additionally, the number of aborted and timed out Tasklets as well as the average latency and load are gathered. Both setups are executed 30 times. The system runs on a Xeon E5345 with 8 cores, Windows Server 2008 Standard, and 6 GB RAM. Each process (the AL and the simulator) has a maximum of 2 GB of RAM. Table 6.1 displays the two setups in this first scenario.

Setup	Use AL	Providers	Consumers	Brokers	Tasklets	Runs
1	Yes	1000	500	2	10000	30
2	No	1000	500	2	10000	30

Table 6.1.: Setups in first Scenario

Evaluation Question 2: In this scenario, the number of entities in the network is increased by the factor 5. This results in 5000 providers and 2500 consumers. The brokers are set to 10 at the start of the system accordingly. This should show how the brokers in the system behave with a high number of registered entities. The evaluation stops after 25000 finished Tasklets. It is referred to this scenario as the second scenario. Both setups are executed 30 times as well. This scenario runs on a Xeon E5-2620 with 64 GB of RAM and Windows Server 2012 R2 Standard. The AL and the simulator have 16 GB of RAM each. Table 6.2 shows the settings of this second scenario.

Setup	Use AL	Providers	Consumers	Brokers	Tasklets	Executions
1	Yes	5000	2500	10	25000	30
2	No	5000	2500	10	25000	30

Table 6.2.: Setups in second Scenario

Evaluation Question 3: Test runs are conducted in the first place to find out feasible feature model sizes. The `KnowledgeFactory` is used to create random models. Since the CFM of the use case has 20 system features and 5 context features these numbers are used for the first model creation process. Then the multiples 40/10, 60/15 and 80/20 (system features/context features) are used. As

only the context features have attributes and one context feature is the parent of all context features, the number of attributes is the number of context features minus one. The tests showed that the latter two configurations are not possible with the current approach. The 60/15 and the 80/20 configurations showed full CPU usage on all cores without any result in multiple attempts and with many hours of runtime. The reason for this is probably a state explosion since permutations are used at one point. More information is given in the discussion section. In order to show that the complexity is also connected with the number of system features, the configuration of 80 system features with 10 context feature models is added to the first two configurations. For each configuration, 10 knowledge input files are created. This results in 30 models. Each model is executed in the ModelTester 10 times resulting in 300 results. Every run has a timeout of 60 seconds for the generation of a result. Execution time in milliseconds per run as well as the number of timeouts per model is gathered. This scenario was executed on a Xeon E3-1240v2 with 32 gigabytes of RAM on Ubuntu Linux 16.04. Table 6.3 shows the intended unusable setups as well as the actually used ones.

Setup	System Features	Context Features	Input files	Runs
1	20	5	10	10
2	40	10	10	10
3	60	15	10	10
4	80	20	10	10

Setup	System Features	Context Features	Input files	Runs
1	20	5	10	10
2	40	10	10	10
3	80	10	10	10

Table 6.3.: Intended and used Setups in third Scenario

Unfortunately, the Sat4J solver was not working in a stable way when executing the adaptation logic inside the productive FESAS runtime. Thus, the evaluation uses the test mode provided by the FESAS Eclipse plugin for executing the adaptation logic. Each part of the evaluation is exported as runnable jar file. The execution of the resulting jar files is orchestrated using batch and shell scripts ac-

cording to the execution platform. The raw data is collected in comma separated value (CSV) files. These are aggregated and processed. The results of these steps are presented in the next section.

6.5.2. Evaluation Results

This section presents the results of the evaluation. A discussion and an interpretation of the results in detail follows.

Evaluation question 1

The first evaluation question is about the capability of the adaptation logic to achieve the goals stated in the feature model. To answer this question, the two setups presented in the previous section are compared. This comparison includes the following measurements: The average provider latency in ms and the average broker load. In contrast, provider fluctuation and the percentage of high-performance requests can be considered as external context values on which the adaptation logic has no influence on. The monitoring data is aggregated in the broker manager the same way as it is done inside the monitoring component for logging purposes. For both monitoring data types the average, the minimum, the maximum, and the standard deviation is calculated for every run.

Figure 6.5 shows the average latency of all 30 runs with and without the adaptation logic. It also depicts the threshold triggering the location policy for latency optimization. This threshold can be seen as feature attribute item in the CFM in Figure 6.2. The graphs of the measurements show that without the adaptation logic the average latency is always above or equal the threshold for triggering the location policy. Using the adaptation logic the goal of achieving a low latency distribution of the TVMs by connecting provider nodes to their nearest broker is met. There is no run where the threshold is met on average. Additionally, it shows that the simulated system using the adaptation logic has in average a 42% lower latency than the simulation system without the adaptation logic.

Table 6.4 shows the complete data set. The average values in connection with the standard deviations in the table imply that the threshold is met multiple times during execution even with the AL. This has to be the case since the AL is reactive. This means the AL tries to resolve an issue once it happens rather than avoiding it proactively.

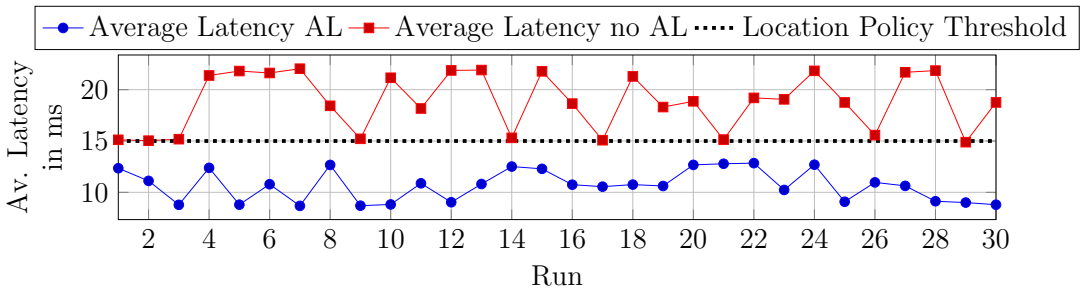

Figure 6.5.: Simulation: Average Latency per Run in first Scenario

	Average		Min		Max		St. Dev.	
Run	AL	No AL	AL	No AL	AL	No AL	AL	No AL
1	12.35	15.12	1	14	23	17	8.25	0.35
2	11.12	15.03	1	14	20	16	7.28	0.21
3	8.78	15.18	1	15	19	18	5.89	0.4
4	12.38	21.38	1	18	23	22	8.28	0.52
5	8.8	21.81	1	18	20	23	5.89	0.45
6	10.79	21.63	1	21	22	23	7.12	0.49
7	8.68	22.04	1	20	19	23	5.81	0.42
8	12.67	18.43	1	17	23	19	8.44	0.5
9	8.7	15.21	1	15	19	17	5.83	0.42
10	8.82	21.16	1	19	20	22	5.82	0.43
11	10.88	18.17	1	17	20	21	7.21	0.5
12	9.03	21.88	1	19	19	23	6.04	0.43
13	10.81	21.91	1	21	22	23	7.22	0.46
14	12.51	15.31	1	14	23	16	8.42	0.47
15	12.29	21.79	1	18	22	23	8.18	0.47
16	10.74	18.64	1	18	20	20	7.08	0.49
17	10.55	15.07	1	14	22	18	7.12	0.3
18	10.75	21.29	1	18	20	22	7.14	0.49
19	10.62	18.31	1	17	20	20	7.05	0.49
20	12.67	18.87	1	16	23	20	8.37	0.49
21	12.78	15.13	1	14	23	18	8.52	0.37
22	12.84	19.2	1	17	23	20	8.52	0.49
23	10.23	19.06	1	17	19	20	6.93	0.46
24	12.69	21.83	1	19	23	23	8.46	0.48
25	9.08	18.76	1	17	18	20	5.96	0.47
26	10.97	15.57	1	15	21	17	7.34	0.5
27	10.63	21.69	1	20	21	23	7.13	0.5
28	9.13	21.86	1	18	19	23	6.11	0.48
29	9	14.89	1	14	20	16	5.98	0.37
30	8.79	18.76	1	17	18	20	5.78	0.47

Table 6.4.: Simulation: Latency in ms per Run in first Scenario

Figure 6.6 shows the average load measurements of the system as a second relevant variable for analysis. As before, the diagram also contains a threshold value which is the start up policy threshold of 60%. It shows that the broker load in the simulation scenario is generally high. As shown in the diagram, this is the case especially for the setup without the adaptation logic. With the two brokers handling the nodes they run at 100% load in average all the time. The adaptation logic achieves to clearly have a lower average load compared to the threshold triggering the start up policy in all runs.

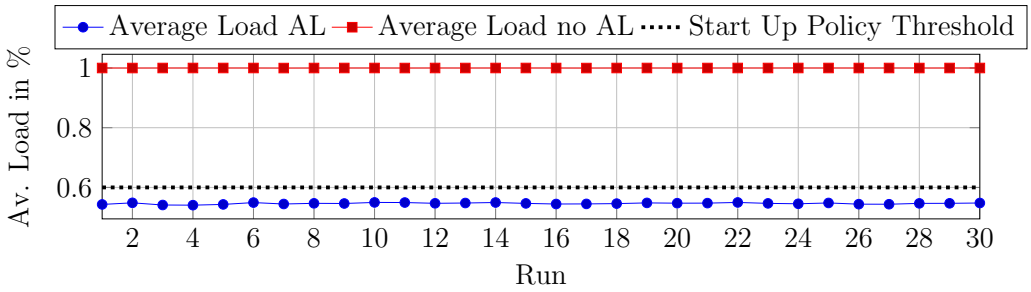

Figure 6.6.: Simulation: Average Load per Run in first Scenario

Table 6.5 shows the complete measurement data of the broker load of the first scenario. Significant results are the stable average values as well as the corresponding standard deviations. However, the standard deviation of the AL setup is measured as relatively high while the setup without AL shows no standard deviation at all.

Figure 6.7 shows the average distribution of active system features. One can see that the adaptation logic triggered reconfigurations due to high latency and high percentage of specialized requests. This is due to the fact that the location and specialization policy are mainly used for the TVM distribution. Thus, the uniform policy, which requires the generic cache list content, is activated only 16 times on average per run. In 41% of the ticks the start up policy is activated. The triggered reconfigurations of the cache list interval features indicate that the fluctuation in the network is low most of the time. The fast interval is activated the fewest of the three cache list intervals. The assumption that the fluctuation is mainly low is also supported by the fact that the cache list length features are almost equally active. Since the cache list content features are directly required by the TVM distribution features, their activation times are the same.

	Average		Min		Max		St. Dev.	
Run	AL	No AL	AL	No AL	AL	No AL	AL	No AL
1	0.54	1	0.02	0.98	1	1	0.40	0
2	0.55	1	0.02	0.93	1	1	0.40	0
3	0.54	1	0.02	0.98	1	1	0.40	0
4	0.54	1	0.02	1.00	1	1	0.40	0
5	0.54	1	0.02	1.00	1	1	0.40	0
6	0.55	1	0.02	0.98	1	1	0.40	0
7	0.54	1	0.02	0.91	1	1	0.39	0
8	0.55	1	0.02	0.96	1	1	0.40	0
9	0.55	1	0.02	1.00	1	1	0.39	0
10	0.55	1	0.02	1.00	1	1	0.40	0
11	0.55	1	0.02	0.96	1	1	0.40	0
12	0.55	1	0.02	0.99	1	1	0.40	0
13	0.55	1	0.02	0.97	1	1	0.40	0
14	0.55	1	0.02	0.99	1	1	0.40	0
15	0.55	1	0.02	1.00	1	1	0.40	0
16	0.54	1	0.02	0.97	1	1	0.40	0
17	0.54	1	0.02	0.92	1	1	0.39	0
18	0.55	1	0.02	0.98	1	1	0.40	0
19	0.55	1	0.02	0.99	1	1	0.39	0
20	0.55	1	0.02	0.94	1	1	0.40	0
21	0.55	1	0.02	0.97	1	1	0.40	0
22	0.55	1	0.02	1.00	1	1	0.40	0
23	0.55	1	0.02	0.98	1	1	0.39	0
24	0.55	1	0.02	1.00	1	1	0.40	0
25	0.55	1	0.02	1.00	1	1	0.40	0
26	0.54	1	0.02	0.96	1	1	0.40	0
27	0.54	1	0.02	0.99	1	1	0.40	0
28	0.55	1	0.02	0.95	1	1	0.40	0
29	0.55	1	0.02	0.99	1	1	0.40	0
30	0.55	1	0.02	1.00	1	1	0.40	0

Table 6.5.: Simulation: Load in % per Run in first Scenario

Besides the monitoring data that is used by the adaptation logic the statistical data of the Tasklet simulation system is also evaluated. Table 6.6 shows the data of the first scenario. It contains the number of timed out and aborted Tasklets. Also, the number of ticks that were needed to finish the predefined number of Tasklets is given. It shows that the number of timed out Tasklets is 196% higher in the setup using the adaptation logic. Additionally, the number of aborted

Tasklets is 13% higher. This results in an average of 33% higher number of ticks for finishing the predefined number of Tasklets.

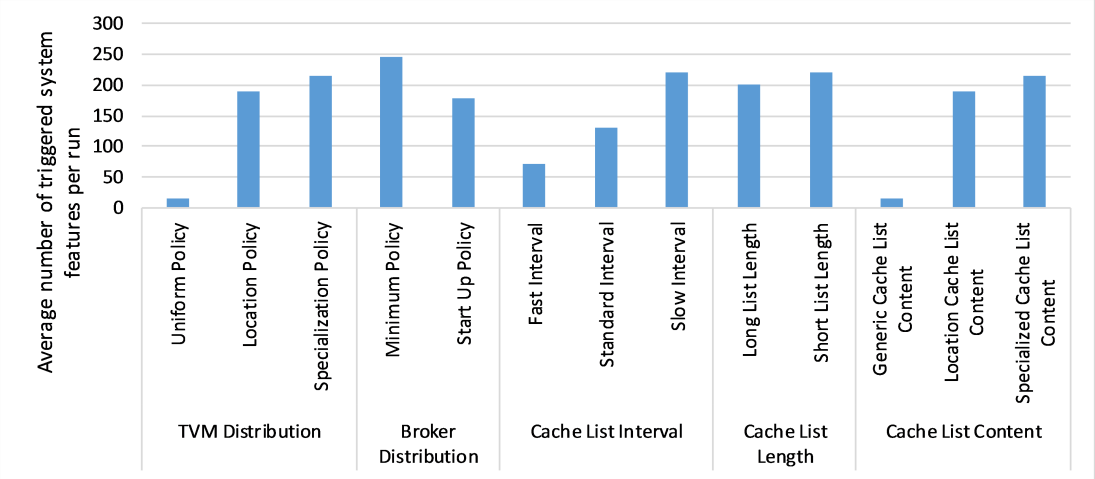

Figure 6.7.: Simulation: Average Number of triggered System Features in first Scenario

	Av. Timeouts	Min T. O.	Max T. O.	St. Dev. T. O.
AL	48371.13	42341	52534	2295.35
No AL	16361.77	15326	17422	521.16
Diff. (%)	+196%	+176%	+202%	+340%

	Av. Aborted	Min A.	Max A.	St. Dev. A.
AL	1618.90	1503	1755	55.24
No AL	1433.97	1296	1573	49.31
Diff. (%)	+13%	+16%	+12%	+12%

	Av. Ticks	Min T.	Max T.	St. Dev. T.
AL	1288.03	1032	1773	203.13
No AL	967.97	916	1035	34.81
Diff. (%)	+33%	+13%	+71%	+484%

Table 6.6.: Simulation: Tasklet Statistics of first Scenario

Figure 6.8 shows the system feature oscillation by looking at the results of one random run in detail. The figure shows the TVM and broker distribution features over time. Concerning the TVM distribution it can be seen that the uniform policy is only activated between the start of the system and the 150th tick. Then the system oscillates between specialization and location policy. The broker distribution continuously oscillates between start up and minimum policy.

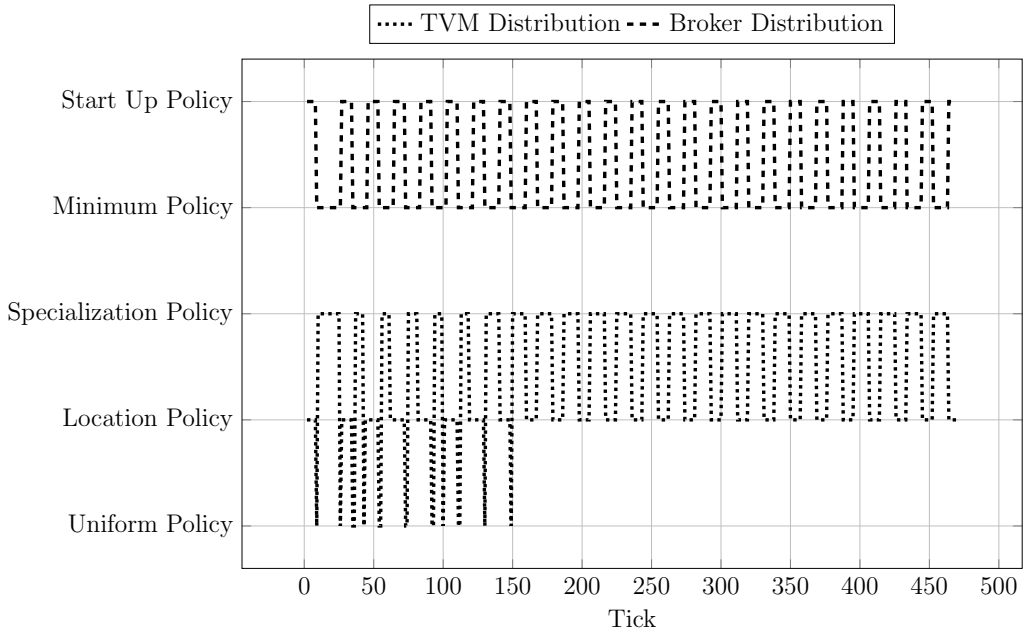

Figure 6.8.: Simulation: Oscillation in one Run in the first Scenario using the AL

Evaluation question 2

For comparison, Figure 6.9 again shows the average latency with and without the adaptation logic in the second scenario. The graphs show similar results compared to the first scenario. One difference is that the latency threshold of 15 ms is met more often than in the smaller first scenario when the adaptation logic is not used.

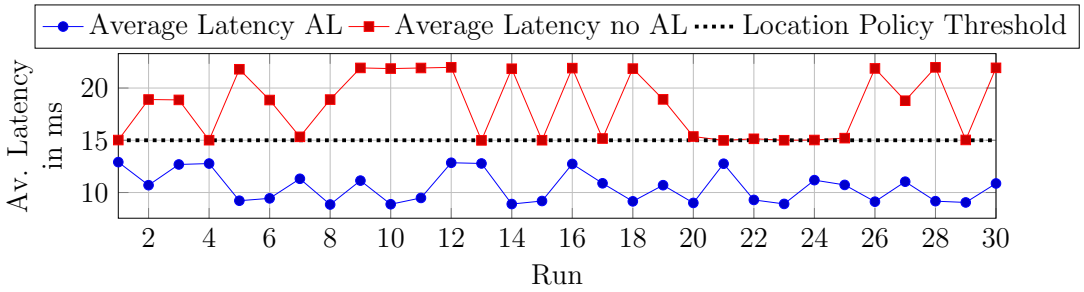

Figure 6.9.: Simulation: Average Latency per Run in second Scenario

Table 6.7 shows the full data set of the latency results. The standard deviation values are very different again. The average values of the setup with the AL are not as stable as the values without it.

	Average		Min		Max		St. Dev.	
Run	AL	No AL	AL	No AL	AL	No AL	AL	No AL
1	12.91	15.02	2	11	23	19	8.07	0.33
2	10.70	18.90	2	14	20	19	6.85	0.48
3	12.68	18.86	1	11	22	19	8.11	0.71
4	12.77	15.01	3	11	22	18	8.15	0.34
5	9.22	21.80	2	12	20	23	5.86	1.36
6	9.44	18.85	2	11	20	20	5.91	0.59
7	11.32	15.33	2	11	20	17	7.05	0.56
8	8.85	18.89	2	11	21	19	5.77	0.72
9	11.14	21.93	2	12	21	23	7.05	0.66
10	8.88	21.86	2	13	19	22	5.76	0.77
11	9.48	21.92	2	12	19	22	5.88	0.75
12	12.85	21.98	1	15	23	23	8.22	0.40
13	12.78	14.99	2	10	23	18	8.12	0.36
14	8.90	21.85	2	12	20	22	5.77	1.03
15	9.19	15.00	2	11	20	18	5.93	0.41
16	12.74	21.90	2	12	23	23	8.15	0.81
17	10.88	15.17	2	11	22	18	6.98	0.51
18	9.16	21.86	2	12	20	22	5.78	0.9
19	10.7	18.91	2	10	20	20	6.86	0.59
20	9.00	15.36	2	12	19	17	5.77	0.55
21	12.76	14.99	2	11	23	17	8.14	0.27
22	9.29	15.14	2	11	19	18	5.98	0.44
23	8.91	15.00	1	10	21	19	5.79	0.43
24	11.19	15.03	2	11	22	18	7.06	0.39
25	10.73	15.21	2	11	20	19	6.87	0.53
26	9.12	21.88	2	10	19	23	5.75	1.09
27	11.04	18.79	2	11	20	19	7.06	0.66
28	9.17	21.98	2	13	20	25	5.86	0.80
29	9.05	15.03	2	10	20	19	5.84	0.40
30	10.87	21.93	2	12	21	25	6.90	0.63

Table 6.7.: Simulation: Latency in ms per Run in second Scenario

Figure 6.10 shows similar load results as in the smaller first scenario. Again, the adaptation logic is able to reach the load goal. Apart from this, there is no other interesting feature present in the figure. For completeness Table 6.8 shows the complete broker load statistics of the second scenario. The results are standard deviation is similar compared to the first scenario.

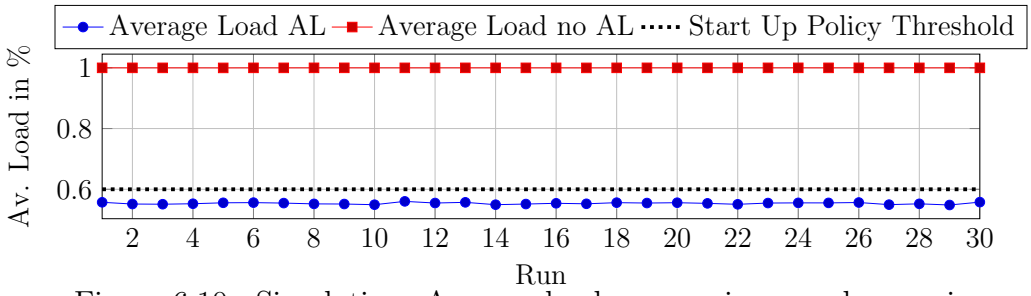

Figure 6.10.: Simulation: Average load per run in second scenario

	Average		Min		Max		St. Dev.	
Run	AL	No AL	AL	No AL	AL	No AL	AL	No AL
1	0.56	1	0.10	0.98	1	1	0.39	0.00
2	0.55	1	0.10	0.98	1	1	0.39	0.00
3	0.55	1	0.08	0.97	1	1	0.39	0.00
4	0.55	1	0.10	0.97	1	1	0.39	0.00
5	0.56	1	0.10	0.96	1	1	0.39	0.00
6	0.56	1	0.10	0.97	1	1	0.39	0.00
7	0.55	1	0.09	0.96	1	1	0.39	0.00
8	0.55	1	0.10	0.99	1	1	0.39	0.00
9	0.55	1	0.10	0.98	1	1	0.39	0.00
10	0.55	1	0.08	0.96	1	1	0.39	0.00
11	0.56	1	0.10	0.96	1	1	0.39	0.00
12	0.55	1	0.06	0.96	1	1	0.39	0.00
13	0.56	1	0.10	0.98	1	1	0.39	0.00
14	0.55	1	0.09	0.98	1	1	0.39	0.00
15	0.55	1	0.08	0.98	1	1	0.39	0.00
16	0.55	1	0.09	0.96	1	1	0.39	0.00
17	0.55	1	0.10	0.96	1	1	0.39	0.00
18	0.56	1	0.10	0.98	1	1	0.39	0.00
19	0.55	1	0.08	0.98	1	1	0.39	0.00
20	0.56	1	0.09	0.97	1	1	0.39	0.00
21	0.55	1	0.09	0.97	1	1	0.39	0.00
22	0.55	1	0.10	0.99	1	1	0.39	0.00
23	0.55	1	0.06	0.97	1	1	0.39	0.00
24	0.56	1	0.08	0.99	1	1	0.39	0.00
25	0.56	1	0.07	0.97	1	1	0.39	0.00
26	0.56	1	0.10	0.98	1	1	0.39	0.00
27	0.55	1	0.10	0.97	1	1	0.39	0.00
28	0.55	1	0.10	0.96	1	1	0.39	0.00
29	0.55	1	0.10	0.98	1	1	0.39	0.00
30	0.56	1	0.08	0.98	1	1	0.39	0.00

Table 6.8.: Simulation: Load in % per Run in second Scenario

Figure 6.11 shows the average processed results of the triggered system features in the large second scenario. The results are very much comparable to the results in the smaller first scenario. It shows that the number of participants in the network did not change the behavior of the adaptation logic. Uniform policy and accordingly the generic cache list content again are the least activated system features. The latency situation is the same as in the smaller first scenario. Especially the TVM distribution feature activations indicate the same results with high load and high demand for specialized Tasklets.

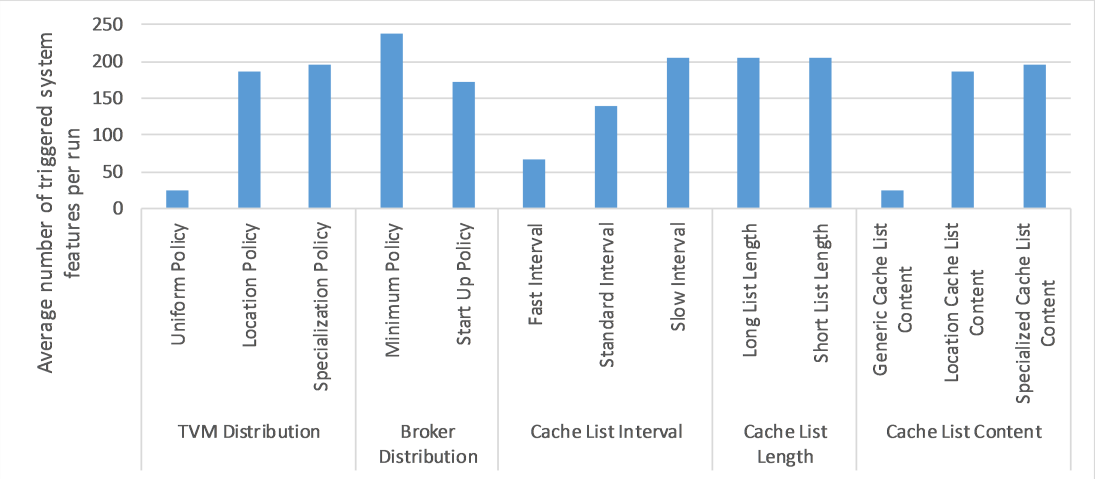

Figure 6.11.: Simulation: Average Number of triggered System Features in second Scenario

For a complete comparison of the first and the second scenario the activation of system features is compared again in one random run of the second scenario. Figure 6.12 shows the same oscillation behavior as Figure 6.8. Again, only in the first 150 the uniform policy gets activated. As the adaptation logic seems to behave in the same way in this scenario as in the first the result was not unpredictable.

Table 6.9 shows the most significant differences. The difference of timed out Tasklets is only a fraction of the value in Table 6.6. The number of aborted Tasklets differs only by 7% in average. Thus, the percentage of overhead of the adaptation logic is not as high as in the first scenario. The most interesting number is the average number of ticks that was needed for finishing the 25000 Tasklets. Using the adaptation logic the number of ticks could be reduced by

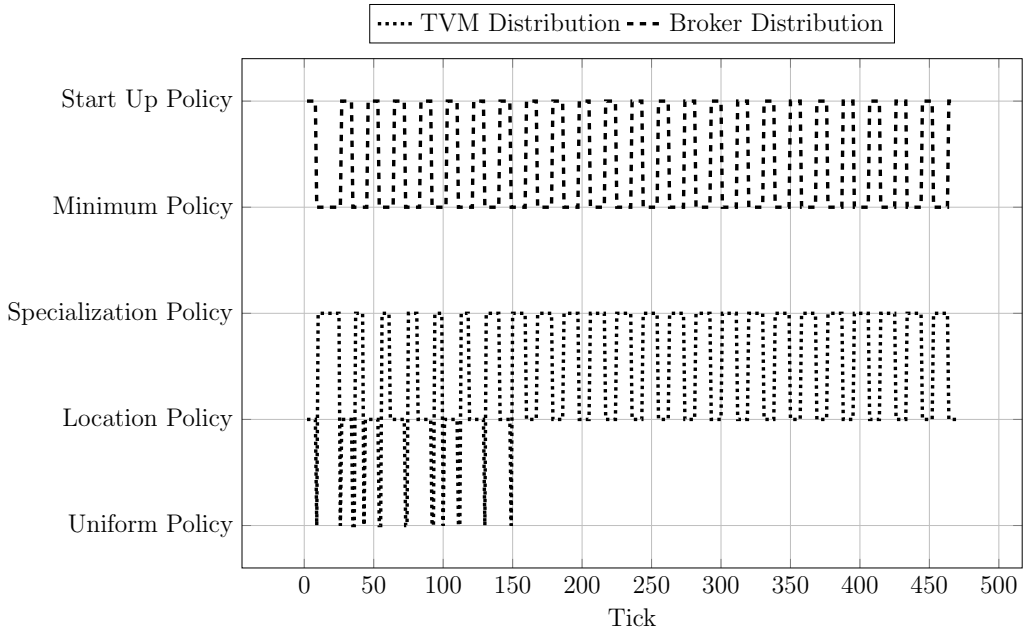

Figure 6.12.: Simulation: Oscillation of Features in one Run in the second Scenario

22% on average.

	Av. Timeouts	Min T. O.	Max T. O.	St. Dev. T. O.
AL	111669.7	106226	120643	3896.40
No AL	87764	81185	98569	3342.93
Diff. (%)	+27%	+31%	+22%	+17%

	Av. Aborted	Min A.	Max A.	St. Dev. A.
AL	3760.53	3536	4023	102.27
No AL	3561.00	3380	3789	89.55
Diff. (%)	+6%	+5%	+6%	+14%

	Av. Ticks	Min T.	Max T.	St. Dev. T.
AL	1244.70	1019	1800	191.22
No AL	1599.87	876	2712	464.62
Diff. (%)	-22%	+16%	-34%	-59%

Table 6.9.: Simulation: Tasklet Statistics of second Scenario

Evaluation question 3

The last evaluation question is concerned with the applicability of the adaptation logic when larger knowledge inputs are used with it. For each setup 10 different knowledge files were created. After the aggregation of the results the following tables show the processed results. Table 6.10 shows the number of feature attribute items. They result from the number of feature attributes. The spreading of the item counts is not very high. The generation of the knowledge files uses random numbers between two and five for the generation of feature attribute items. Thus, it basically shows exactly this range. Still, this information is important as a high number of feature attribute items results in more complex processing in the SAT solver. This is especially the case if conflicts occur.

System F.	Context F.	Min FAI	Max FAI	St. Dev. FAI	Average FAI
20	5	9	19	3.09	14.0
40	10	25	40	4.28	29.5
80	10	26	35	3.00	30.4

Table 6.10.: Model Testing: Aggregated number of Feature Attribute Items (FAI)

Table 6.11 shows results for the runtime in milliseconds per knowledge configuration. Again, minimum, maximum, standard deviation and average are displayed. The numbers grow very fast starting with the small configuration reaching the larger configuration. Increasing values are also viewable in Table 6.12. The table contains the number of timeouts of each of the three setups. The number of timeouts is 0 for the first two setups and increases very fast to over 90 in the third setup. A timeout happened in case more than 60 seconds were needed to select a configuration. In this case the complete Java VM is restarted.

System F.	Context F.	Min ms	Max ms	St. Dev. ms	Average ms
20	5	8	161	18.15	19.72
40	10	15	22510	3012.92	1677.04
80	10	83	55646	12497.90	7323.99

Table 6.11.: Model Testing: Aggregated Runtime in ms

For a more detailed view onto the different knowledge files for the three setups Tables 6.13, 6.14, and 6.15 are depicted as well. They display the results of

Sys. F.	Ctx F.	Timeouts
20	5	0
40	10	0
80	10	94

Table 6.12.: Model Testing: Aggregated Number of Timeouts

the runtime in ms per knowledge file of each setup. Table 6.13 shows that in this small setup a configuration is mostly selected relatively fast. In this setup executing knowledge file 5 results in the highest runtimes. The same applies to knowledge file 9 in Table 6.14. A higher number of feature attribute items does not necessarily increase the runtime within the same setup. For example, the knowledge files 1, 5, and 6 in Table 6.15 share 29 as number of attribute items. Still, the runtime results are very different. Since the 80/10 setup was the only setup with timeouts, Table 6.16 shows the timeouts aggregated per model. The range of timeouts reaches from 29 to even 0 in this setup.

Know. File	FAI	Min ms	Max ms	St. Dev. ms	Average ms
1	9	11	26	6.03	18.8
2	15	11	65	16.69	27.8
3	10	10	51	11.95	19.1
4	13	8	29	7.04	15.6
5	14	11	161	45.55	40.2
6	14	8	19	3.72	12.6
7	19	9	50	14.27	20.1
8	15	10	37	9.00	16.7
9	13	8	16	2.50	11.7
10	18	10	22	3.41	14.6

Table 6.13.: Model Testing: Runtime Results in the 20/5 Setup

Know. File	FAI	Min ms	Max ms	St. Dev. ms	Average ms
1	32	270	2212	472.30	1351.7
2	40	16	2611	886.01	1086.2
3	27	15	1881	719.39	1037.2
4	25	21	1512	642.60	948.6
5	27	28	1427	530.92	939.2
6	32	42	1605	563.68	1045
7	31	39	4573	1398.38	1597.6
8	27	23	1872	472.11	1221.2
9	30	96	22510	7832.58	6813.3
10	29	31	1854	802.58	730.4

Table 6.14.: Model Testing: Runtime Results in the 40/10 Setup

Know. File	FAI	Min ms	Max ms	St. Dev. ms	Average ms
1	29	338	40613	15197.21	20941.7
2	33	1554	8492	2836.34	2960.1
3	26	1401	1734	90.58	1538.5
4	35	305	36459	11204.41	4980.5
5	29	1034	55646	17800.76	11839.8
6	29	83	4631	1204.22	1686.8
7	32	1549	4366	870.61	1894.5
8	34	1490	47325	18348.91	19537.3
9	27	1674	7186	1584.38	2863.7
10	30	265	38671	11843.85	4997

Table 6.15.: Model Testing: Runtime Results in the 80/10 Setup

File	1	2	3	4	5	6	7	8	9	10
Timeouts	29	1	1	15	20	5	7	1	0	15

Table 6.16.: Model Testing: Timeouts in the 80/10 Setup

6.6. Discussion

This section discusses the results of the previous section and their support for the evaluation questions. The discussion is done sequentially, i.e. question by question. Then a qualitative evaluation discussion of the CFM-based AL follows. Therefore, the CFM-based AL is characterized according to additional DSPL guidelines [BBD16] state in their work. The last part of this section describes limitations of the evaluation.

6.6.1. Interpretation of the results

Evaluation question 1

The results presented in the last section support that the latency and broker load goals of the system are achieved when using the adaptation logic. The latency results show a relatively high gap between the average values and the thresholds. Referring to Figure 6.2 showing the CFM of the use case the FAI with the latency range 10-15 ms might be the reason for this. Neither does it trigger the minimum policy nor the location policy. This means the system runs the location policy longer when the latency lowers over time. This explains the gap between the average latency result when the AL manages the system and the location policy threshold. Apart from these positive results Table 6.6 shows that the AL increases the number of timed out Tasklets significantly. This increase may happen due to the fact that providers receiving a provider request which is too complex for them forward this request. In the AL setup it is easily possible that a broker that received the forwarded requests is stopped immediately. Thus, it is possible that a request is forwarded to a broker that gets removed quickly. This triggers the timeout in the consumer. The other interesting metric is the number of needed ticks. Without the adaptation logic significantly fewer ticks are needed to achieve the number of Tasklets which had to be finished. In the AL setup each time a TVM or broker distribution feature gets activated it sends out a provider status request to all known providers of all brokers. Providers answer this with their provider status. Thus, a lot of additional messages are sent through the system. Also, the number of brokers and the distribution of providers is changing all the time. This results in an overall high management overhead. The last figure in the results of the first scenario is concerned with oscillation. Figure 6.8 shows that

the system often switches back and forth between two configurations, which can be a problem in real world scenarios. Here the behavior of the broker distribution features is reasonable since the minimum policy stops unneeded brokers, while the start up policy starts new ones. The TVM distribution is not a binary xor group. However, the oscillation between specialization and location policy supports that the number of high-performance requests and the overall latency is generally high in the simulation. This is supported by Figure 6.7.

Evaluation question 2

Looking at the results in Figure 6.9 there is no big difference compared to the lower participant scenario results used to answer evaluation question 1. Still, it shows that the configurations generated by the adaptation logic fulfill the goal of a latency lower than 15 ms (as defined in Figure 6.2). Overall there is no big difference compared to 6.5. Figure 6.10 shows the average load in the system. It supports that the adaptation logic fulfills the goal to achieve an average network load which is lower than 60% (as defined in Figure 6.2). This figure also displays no big difference compared to 6.10. Figure 6.11 shows the distribution of activated system features. These highly aggregated numbers show no difference to the average system feature activations in the smaller scenario (cf. to Figure 6.7). As the broker manager is the managed resource in the use case it does not change with the number of nodes in the network. This results in the same behavior as in the first scenario. It's the same with the graphs showing the average latency and load. Just like before, the broker manager is the interface to the actual brokers. Nothing has changed in these terms. The most interesting differences can be seen in Table 6.9. The increase of timeouts that is caused by the adaptation logic is much lower compared to Table 6.6 showing the results of the first scenario. As the standard deviation is not very high the numbers are also very certain. The number of aborted Tasklets is not very different between the runs with and without the adaptation logic again. However, percentage-wise this difference is also lower than in the first scenario. Apart from the negative effect of having higher numbers of timed out and aborted Tasklets, the number of average ticks needed to finish the goal of 25000 finished Tasklets decreased by 22% on average. Still, it is important to note that the standard deviation from the results running the second scenario without adaptation logic is relatively high. At the same time, the minimum value is lower without the AL just as the maximum value is a lot

higher. In summary, it can be said that the adaptation logic is perfectly capable of handling a high number of entities running in the Tasklet system. Since the managed resource does not change at all with different numbers of entities this is not very surprising. In both scenarios the managed resource is the same broker manager. Still, the results imply that the overhead of the adaptation logic is not as high as in scenarios with less entities. This can be seen in the percentage-wise lower increase of timed out Tasklets compared to the first scenario. The biggest achievement is the 22% lower number of ticks that was needed by the Tasklet system finishing 25000 Tasklets. This signifies that the system performs better in large Tasklet environments using the adaptation logic while in smaller installations the AL does not improve at all.

Evaluation question 3

Table 6.10 shows the number of feature attribute items. What is interesting is the comparison of the average of attribute items between the 40/10 and the 80/10 configurations. The average item count in these setups are almost the same. In fact, this should be the case as both setups use the same number of context features. This results in the same number of context feature attributes. Table 6.11 shows that in terms of the minimum runtime in milliseconds the first and second configurations have a very low minimum while the third one has a much higher minimum. Doubling the number of context features increases the minimum runtime by the factor 5.5. The maximums in combination with the standard deviation and the average value show that they are outliers. The typical average value is much lower than this. Looking at the average values of the second and third configuration they show that selecting a system feature configuration using the AL takes up to multiple seconds most of the time. In general, this can be an obstacle for systems with the need of very fast reaction time. This is especially notable as the number of features in these three setups is not particularly high.

Table 6.12 supports that the larger models have a higher complexity. Thus, there may be multiple runs in the third setup where the adaptation logic possibly creates a result in more than 60 seconds of time.

As these tables only show the highly aggregated results the details per run are also important. Table 6.13 shows the results of the smallest knowledge file that is comparable to the use case knowledge. At first sight, there is no direct corre-

lation between high attribute item counts and runtimes. Comparing knowledge file 5 and 6 shows that even though they have the same number of FAIs, their runtime results are very different. Knowledge file 5 has the highest runtime results compared to all other files while file 6 is the second fastest on average. This shows that other aspects such as the number and complexity of constraints as well as priority and cost values for resolving conflicts are important to the runtime. Selecting the cheapest configuration also takes at least some time. Having a similar case in Table 6.14 with knowledge file 1 and 6 there is not such a big difference. Finding out correlations in this domain is an item for future work. The last two tables for question 3 show the results of the 80/10 setup. According to Table 6.15 knowledge files 1, 5, and 8 are the most complex knowledge files. The timeouts results in Table 6.16 only support this for file 1 and 5. Another interesting point is that knowledge file 3 has a very low standard deviation. Also, it has the lowest number of FAIs. This suggests that in this setup there is a correlation between the number of FAIs and the standard deviation in runtime. In general, the standard deviation is very high in this setup for many knowledge files. Thus, this shows that the aggregation of this data hides the big differences between the different model files.

Concerning the infeasibility of the 60/15 and the 80/20 setups they both showed a state explosion during the first tests. As already mentioned permutations of the FAIs are used in one step. This makes it impossible to run large configurations with many FAIs at the moment. The reason 15 and 20 context features are too many for this approach comes from the number of FAIs. 15 context features result in 14 feature attributes in the current evaluation setting. These result in 14 times 5 FAIs in the worst case. 70! FAIs are too many for analysis. 20 context features are even worse. This is also one limitation of this thesis. They are presented in the last section of this chapter.

Finally, the execution time increases very fast with the number of features in the system. Explicit correlation results cannot be stated using these results. Runtimes with the length of multiple seconds may be a problem in very fast execution environments where decisions have to be made very quickly.

6.6.2. Characterization based on DSPL Guidelines

After the discussion of the qualitative evaluation results this section characterizes the adaptation logic. In fact, it relates the DSPL guidelines from [BBD16] to the proposed approach.

Monitoring and Analysis Design
As in this approach, the context model is organized like features, it is a property-set approach. Using ontologies could make it possible to perform more complex and semantic reasoning on the context. However, this requires to build a whole ontology on your own or to utilize an existing one. According to [BBD16], using ontologies is especially beneficial in multi-agent environments as ontologies can easily be shared and the reasoning approaches are generic enough to enable reasoning for multiple agents. Since this is a single agent environment and the planning is entirely based on the CFM, this property-set approach works fine.

A property-set approach can easily be used with a rule base as context reasoning model. This rule base is represented by the constraints in the context feature model. Here, it is not possible to use any kind of specific query language.

Context sensing depends on the possible interfaces for context retrieval of the managed resource. If the managed resource supports the publish-subscribe pattern, notification on context changes is possible. Otherwise, the adaptation logic has to poll periodically for new context information. As this dimension is use case dependent, it cannot be specified up front.

Planner Design
Regarding the variability space model this approach obviously uses a feature model for representing the variability. This is the most powerful method of this dimension for representing variability in a structured way.

The planning model depends on the planning type [BBD16]. Thus, planning model and type are described together. The CFM constraints together with the cross-tree constraints represent a rule base. They specify thresholds as well as requirements on the basis of context features. Thus, they perfectly define ECA rules: the event is the allocation of a context attribute based on context information, the condition is the exceedance of a threshold, and the action is to switch specified features on or off.

Goal-based and utility-based approaches are most effective when the configuration space is not set up completely at design time, which is not the case here as the CFM is set up a priori. A utility-based approach is mainly used for self-optimization, which is not the focus of this work. However, it may be possible to work with some global utility value for evaluating the performance of a CFM-based system by giving features weights and combining them to one utility value.

Considering planning, which takes place on the basis of feature models, the planning level are the features themselves. Each activated feature may require other features to be enabled or disabled. The actual architecture and implementation is separated from the planner and reconfiguration decisions. In the end, the planner selects a single configuration the actual system has to implement. As the feature model is part of the adaptation logic here and the feature implementation is part of the managed resource, the planning level is clearly feature-based.

Concerning the transformation dimension, the feature model is set up to represent the architecture of the software system directly meaning that one feature represents a certain component or class. This means the transformation is based on the direct link approach. Aspect model weaving may also be applicable. However, the overhead to manage the aspects is very likely to be higher compared to the direct link approach ([BBD16]). Concerning transformation rules they often require a running SAT solver at runtime also increasing the overhead. Here, the SAT solver is used in the planning component only to select a valid and good configuration.

Execution Design

The architectural model determines the complexity of reconfigurations. Thus, it is supposed to offer a good level of abstraction that facilitates the ease of reconfigurations. The method used here represents the architecture by the feature model. Thus, there is no additional architectural model for representing the system.

The architecture style determines the dynamism of the reconfigurations. For one system functionality, a group of sub-features may be able to fulfill this functionality in different ways. As deploying a configuration means to replace certain system features completely, the approach presented is component-based. Subsequently, the variation entities are components as well. The runtime reconfiguration is

described by the component model dimension as well. The reason for this is that no dynamic aspect weaving is used in this approach.

Additionally, the authors of [BBD16] mention the importance of extensibility of the middleware and the component model. The component model should be able to easily add sensors to the system for adding sensor values dynamically. These requirements are met by the use of FESAS as development framework. FESAS offers an extensible skeleton with the possibility to easily replace complete components. Also, it offers the needed sensing capabilities.

6.6.3. Limitations

This section presents limitations of the CFM-based adaptation logic approach as well as limitations of the simulation-based approach for evaluation.

The underlying metamodel depicted in Figure 5.2 is limited in terms of the number of feature attributes. In the current approach only one single feature attribute is possible per feature. In general, multiple attributes would be possible per feature. The number of possible models in the CFM of the use case is also rather small. Yet, the results of evaluation question 3 indicate problems with larger models that have to be solved.

The CFM-based adaptation logic is limited to SAT solving at the moment. Especially when integer or floating point feature attributes are added, a CSP solver component can help. Also, the conflict solving and feature selection on the foundation of priority and cost values is very simple at the moment. There is no cost function present in the current approach as each system feature has a fixed cost predefined at design time. A cost function may be able to dynamically calculate the system cost per feature or even per complete or partial configuration. Another point is the idea of having a stop command for the monitoring component. On the one hand it is possible to control the adaptation logic from the managed resource this way, on the other hand this should not be needed. The adaptation logic should be able to determine on its own how the monitoring data should be handled.

There are multiple limitations concerning the simulation approach presented here. A simulator cannot directly represent the behavior of the real system. In fact,

since the entities in the system are not really independent and their methods are called entity-by-entity by the simulator, their asynchronous operation is not represented in this simulation. Also, in a real world scenario, the monitoring data may not be complete all the time. It could be the case that the adaptation logic has to decide using incomplete information. This cannot happen here. However, the model supports the possibility of partial knowledge already. Another limitation is caused by some direct method calls which were needed because of the architecture that was implemented in this simulation. In a real system, everything has to be a message rather than a method call.

For all evaluation components including the model checker multi-threading is a big issue. Only one core is used. Thus, the performance of the components is completely dependent on the single core performance of the executing system. This was especially an issue for answering question 1 and 2. These scenarios were executed on multi-core CPUs with low clock per core. The server used in scenario 2 had 24 cores and only one was used at a time. Thus, multi-core operation would result in a much higher performance.

Another limitation comes from the Sat4J library that was used for SAT solving. In case of a conflict, the library provides a method for determining the actual literals that are in conflict. However, this method results in being not reliable. The resolution here is to try all permutations of the context feature attribute items in case of a conflict for determining the minimum features in conflict. Since the number of permutations results in the faculty of the number of FAIs, this is an enormous bottleneck. This is the reason that scenario 2 required 16 GB of RAM per Java VM. Otherwise, the Java VM would crash with a low memory exception.

These limitations imply that there are a lot of possibilities for further improvements and for further research. The following chapter finalizes this work. It summarizes the findings and eventually finishes this thesis by describing opportunities for further research.

7. Summary and Further Research

In the first section there is an outline of the key findings of every chapter followed by the conclusions of this thesis. As the results of this work cannot be complete there is an outline of potential future research.

7.1. Summary

Chapter 2 introduces the fundamentals of self-adaptive systems, software product line and dynamic software product lines. As one result it was clear that the adaptation logic approach of this thesis should be external. Also, the context-aware DSPL techniques should be used inside the adaptation logic. This is the most powerful DSPL technique. Then related work in the field of dynamic software product line approaches is presented. This includes the adaptation and DSPL taxonomies of [BBD16]. The goal of this thesis is to provide a generic way for specifying variability for all kinds of systems without the need to learn a special modeling technique. Thus, the generic context-aware feature modeling approach by [SLR13] is used as the foundation of the DSPL adaptation logic approach of this thesis. Subsequently, the approach of this thesis is introduced and characterized. The implementation uses Java and FESAS. The Tasklet distributed computing system is the use case for the qualitative evaluation of this thesis. A simulated system is used for this purpose. The evaluation shows multiple properties of the adaptation logic. One important finding is that the overhead of the adaptation logic leads to bad results in small Tasklet scenarios. The reason for this is that the improvements in such a scenario do not compensate for the performance degradations of the overhead. However, the large Tasklet scenario is finished significantly faster employing the adaptation logic. One problem is that no multi-core operation is possible at the moment. This leads to the last section elaborating on possibilities for further research.

7.2. Further Research

As already mentioned, multi-core support would help improve the performance. Using only one core primarily impairs performance on multi-core systems with low CPU clock. Another possibility is to use a different SAT solver than Sat4J. In benchmarks of the so-called SAT competition, the Sat4J does not compete very well compared to other solvers [BBJS15]. It was selected because it was the only SAT solver available in Java. It may be possible to write the CNF to disc and use a C-based faster solver instead. Additionally, profiling of each method inside the adaptation logic could lead to performance bottlenecks as well. As mentioned earlier, instead of only supporting SAT representations additional CSP and SMT solvers may also be integrated into the adaptation logic. Apart from the current support for reactive operation, proactive capabilities should be added as well.

Concerning the implementation of the metamodel, there are possibilities for further research as well. The metamodel can be expanded further, e.g., to support multiple feature attributes per feature. In addition, the supported attribute types and constraints can be extended.

Regarding the simulation system, the correlations between feature count, attributes, and constraints have to be explicitly explored. Also, it would be interesting to have multiple broker management instances including the capability to manage each broker's configuration separately. An additional use case could also help extend the knowledge on correlations between number of features in the model and runtimes. This could include qualitative context features such as stability or reliability. The value ranges of the already context feature attributes could also be optimized for best suitability. Finally, using a testbed rather than a simulator should be the goal eventually.

This outlook for further research concludes this thesis. The presented adaptation logic approach combining DSPL-based context feature models with a MAPE-K cycle is a good starting point for further investigation. This will result in new and improved AL approaches based on context feature models in the future.

Bibliography

[AAL10] Nadeem Abbas, Jesper Andersson, and Welf Löwe. Autonomic software product lines (aspl). In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, ECSA '10, pages 324–331. ACM, 2010.

[ACF+09] Mathieu Acher, Philippe Collet, Franck Fleurey, Philippe Lahire, Sabine Moisan, and Jean Paul Rigault. Modeling context and dynamic adaptations with feature models. In *CEUR Workshop Proceedings*, volume 509, pages 89–98, 2009.

[BBD16] Mahdi Bashari, Ebrahim Bagheri, and Weichang Du. Dynamic software product line engineering: a reference framework. Technical report, 2016.

[BBJS15] Adrian Balint, Anton Belov, Matti Järvisalo, and Carsten Sinz. Overview and analysis of the sat challenge 2012 solver competition. *Artificial Intelligence*, 223:120 – 155, 2015.

[BCL+06] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean Bernard Stefani. The FRACTAL component model and its support in Java. *Software - Practice and Experience*, 36(11-12):1257–1284, 2006.

[BDG+13] Yuriy Brun, Ron Desmarais, Kurt Geihs, Marin Litoiu, Antonia Lopes, Mary Shaw, and Michael Smit. *A Design Space for Self-Adaptive Systems*, pages 33–50. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[BDMSG+09] Yuriy Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi Müller, Mauro Pezzè, and Mary Shaw. *Engineering Self-Adaptive Systems through Feedback Loops*, pages 48–70. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[Ben04] Hachemi Bennaceur. A comparison between SAT and CSP techniques. *Constraints*, 9(2):123–138, 2004.

[BGF+08] Nelly Bencomo, Paul Grace, Carlos Flores, Danny Hughes, and Gordon Blair. Genie: Supporting the Model Driven Development of Reflective, Component-based Adaptive Systems. *Proceedings of the 13th international conference on Software engineering - ICSE '08*, pages 811–814, 2008.

[BGL+12] Luciano Baresi, Sam Guinea, Pasquale Liliana, Mike Hinchey, Sooyong Park, and Klaus Schmid. Service-Oriented Dynamic Software Product Lines. *Computer*, 45(10):42–48, 2012.

[BHA12] Nelly Bencomo, Svein Hallsteinsen, and Eduardo Santana De Almeida. A view of the dynamic software product line landscape. *Computer*, 45(10):36–41, 2012.

[BPSM+98] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (xml). https://www.w3.org/TR/1998/REC-xml-19980210, 1998. Accessed: 08.01.2017.

[BSBG08] Nelly Bencomo, Peter Sawyer, Gordon S. Blair, and Paul Grace. Dynamically Adaptive Systems are Product Lines too: Using Model-Driven Techniques to Capture Dynamic Variability of Adaptive Systems. In *SPLC (2)*, pages 23–32, 2008.

[BSGR03] Christian Becker, Gregor Schiele, Holger Gubbels, and Kurt Rothermel. Base - a micro-broker-based middleware for pervasive computing. In *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications, 2003. (PerCom 2003).*, pages 443–451, March 2003.

[BSRC10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010.

[BSST09] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability Modulo Theories. In *Handbook of satisfiability*, volume 185. 2009.

[BTRC05] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Automated Reasoning on Feature Models. *LNCS Advanced Information Systems Engineering 17th International Conference CAiSE 2005*, 01:491–503, 2005.

[CAA09] Lianping Chen, Muhammad Ali Babar, and Nour Ali. Variability management in software product lines: a systematic review. *Proceedings of the 13th International Software Product Line Conference*, pages 81–90, 2009.

[CBG+08] Geoff Coulson, Gordon Blair, Paul Grace, Francois Taiani, Ackbar Joolia, Kevin Lee, Jo Ueyama, and Thirunavukkarasu Sivaharan. A generic component model for building systems software. *ACM Trans. Comput. Syst.*, 26(1):1:1–1:42, March 2008.

[CBT+14] Rafael Capilla, Jan Bosch, Pablo Trinidad, Antonio Ruiz-Cortés, and Mike Hinchey. An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry. *Journal of Systems and Software*, 91(1):3–23, 2014.

[CFP08] Carlos Cetina, Joan Fons, and Vicente Pelechano. Applying software product lines to build autonomic pervasive systems. *Proceedings - 12th International Software Product Line Conference, SPLC 2008*, (ii):117–126, 2008.

[CHE04] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. *Staged Configuration Using Feature Models*, pages 266–283. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[CHSL11] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. Symbolic model checking of software product lines. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 321–330, New York, NY, USA, 2011. ACM.

[Cro06] Douglas Crockford. The application/json media type for javascript object notation (json). https://tools.ietf.org/html/rfc4627, 2006. Accessed: 23.10.2016.

[Dav87] Stanley M Davis. *Future Perfect*. 1987.

[DDD+13] Jackson Raniel F Da Silva, Francisco Airton P Da Silva, Leandro M. Do Nascimento, Dhiego A O Martins, and Vinicius C. Garcia. The dynamic aspects of product derivation in DSPL: A systematic literature review. *Proceedings of the 2013 IEEE 14th International Conference on Information Reuse and Integration, IEEE IRI 2013*, pages 466–473, 2013.

[ESK+17] Janick Edinger, Dominik Schäfer, Christian Krupitzer, Vaskar Raychoudhury, and Christian Becker. Fault-Avoidance Strategies for Context-Aware Schedulers in Pervasive Computing Systems. *Proceedings of the 15th IEEE International Conference on Pervasive Computing and Communications, 2017. (PerCom 2017).*, page To be published, 2017.

[FHS+06] Jacqueline Floch, Svein Hallsteinsen, Erlend Stav, Frank Eliassen, Ketil Lund, and Eli Gjørven. Using architecture models for runtime adaptability. *IEEE Software*, 23(2):62–70, 2006.

[Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[GFD98] M.L. Griss, J. Favaro, and M. D'Alessandro. Integrating feature modeling with the RSEB. *Proceedings. Fifth International Conference on Software Reuse (Cat. No.98TB100203)*, pages 76–85, 1998.

[GH04] Hassan Gomaa and Mohamed Hussein. Dynamic Software Reconfiguration in Software Product Families. *Software Product-Family Engineering*, 3014(iii):435–444, 2004.

[GJ83] Joel D Goldhar and Mariann Jelinek. Plan for economies of scope. *Harvard Business Review*, (November):141–149, 1983.

[GS03] Jack Greenfield and Keith Short. Software factories: assembling applications with patterns, models, frameworks and tools. *Systems, Languages, and Applications*, pages 16–27, 2003.

[HHSS08] S. Hallsteinsen, M. Hinchey, Sooyong Park, and K. Schmid. Dynamic Software Product Lines. *Computer*, 41(4):93–95, 2008.

[Inc06] Object Management Group Inc. Common variability language. http://www.omgwiki.org/variability/doku.php, 2006. Accessed: 02.08.2016.

[KC03] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, Jan 2003.

[KCH+90] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.

[KKL+98] Kyo C Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. FORM: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5:143–168, 1998.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. *ECOOP'97 — Object-Oriented Programming*, 1241/1997:220–242, 1997.

[KRB+16] Christian Krupitzer, Felix Maximilian Roth, Christian Becker, Markus Weckesser, Malte Lochau, and Andy Sch. Fesas ide: An integrated development environment for autonomic computing. In *2016 IEEE International Conference on Autonomic Computing (ICAC)*, pages 15–24, July 2016.

[KRP16] Christian Krupitzer, Felix Maximilian Roth, and Martin Pfannemüller. Comparison of approaches for self-improvement in self-adaptive systems. In *2016 IEEE International Conference on Autonomic Computing (ICAC)*, pages 308–314, July 2016.

[KRVB15] Christian Krupitzer, Felix Maximilian Roth, Sebastian Vansyckel, and Christian Becker. Towards reusability in autonomic computing. *Proceedings - IEEE International Conference on Autonomic Computing, ICAC 2015*, pages 115–120, 2015.

[KVB13] Christian Krupitzer, Sebastian Vansyckel, and Christian Becker. Fesas: Towards a framework for engineering self-adaptive systems.

In *2013 IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems*, pages 263–264, Sept 2013.

[Lad01] Robert Laddaga. Active software. In *Self-Adaptive Software: First International Workshop, IWSAS 2000 Oxford, UK, April 17–19, 2000 Revised Papers*, pages 11–26, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[LBL08] Daniel Le Berre and Inês Lynce. Csp2sat4j: a simple csp to sat translator. *Proceedings of the 2nd International CSP Solver Competition*, pages 43–54, 2008.

[LP10] Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2 system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(2010):59–64, 2010.

[LRS03] Robert Laddaga, Paul Robertson, and Howie Shrobe. Introduction to self-adaptive software: Applications. pages 1–5, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[MBJ+09] Brice Morin, Olivier Barais, Jean-Marc Jezequel, Franck Fleurey, and Arnor Solberg. Models@ Run.time to Support Dynamic Adaptation. *Computer*, 42(10):44–51, 2009.

[MSKC04] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. Composing adaptive software. *Computer*, 37(7):56–64, July 2004.

[OGT+99] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, May 1999.

[PAG06] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. *Semantics and Complexity of SPARQL*, pages 30–43. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[PBD09] Carlos Parra, Xavier Blanc, and Laurence Duchien. Context awareness for dynamic service-oriented product lines. *13th International Software Product Line Conference*, pages 131–140, 2009.

[PBV05] K. Pohl, G. Böckle, and F. Van Der Linden. *Software Product Line Engineering. Foundations, Principles, and Techniques*, volume 49. 2005.

[Pet15] Justyna Petke. *Bridging Constraint Satisfaction and Boolean Satisfiability*. Springer, 2015.

[RBSP02] M Riebisch, K Böllert, D Streitferdt, and I Philippow. Extending Feature Diagrams with UML Multiplicities. *6th World Conference on Integrated Design & Process Technology*, pages 1–7, 2002.

[RN09] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 3rd edition, 2009.

[Sat09] Satcompetition.org. Cnf file format. http://www.satcompetition.org/2009/format-benchmarks2009.html, 2009. Accessed: 02.08.2016.

[SEI] Carnegie Mellon University Software Engineering Institute. Software product lines. https://www.sei.cmu.edu/productlines/. Accessed: 02.08.2016.

[SHTB06] Pierre Yves Schobbens, Patrick Heymans, Jean Christophe Trigaux, and Yves Bontemps. Feature Diagrams: A survey and a formal semantics. *Proceedings of the IEEE International Conference on Requirements Engineering*, pages 136–145, 2006.

[SLR13] Karsten Saller, Malte Lochau, and Ingo Reimund. Context-aware DSPLs: model-based runtime adaptation for resource-constrained systems. *Proceedings of the 17th International Software Product Line Conference co-located workshops*, pages 106–113, 2013.

[ST09] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):14:1–14:42, May 2009.

[W3C04] World Wide Web Consortium W3C. Owl web ontology language - semantics and abstract syntax. https://www.w3.org/TR/owl-semantics/, 2004. Accessed: 21.11.2016.

[W3C08] World Wide Web Consortium W3C. Sparql query language for rdf.

https://www.w3.org/TR/rdf-sparql-query/, 2008. Accessed: 21.11.2016.

[Wal00] Toby Walsh. SAT v CSP. In *CP '02 Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming*, pages 441–456, 2000.

Appendix

A. Related Work in DSPL Approaches

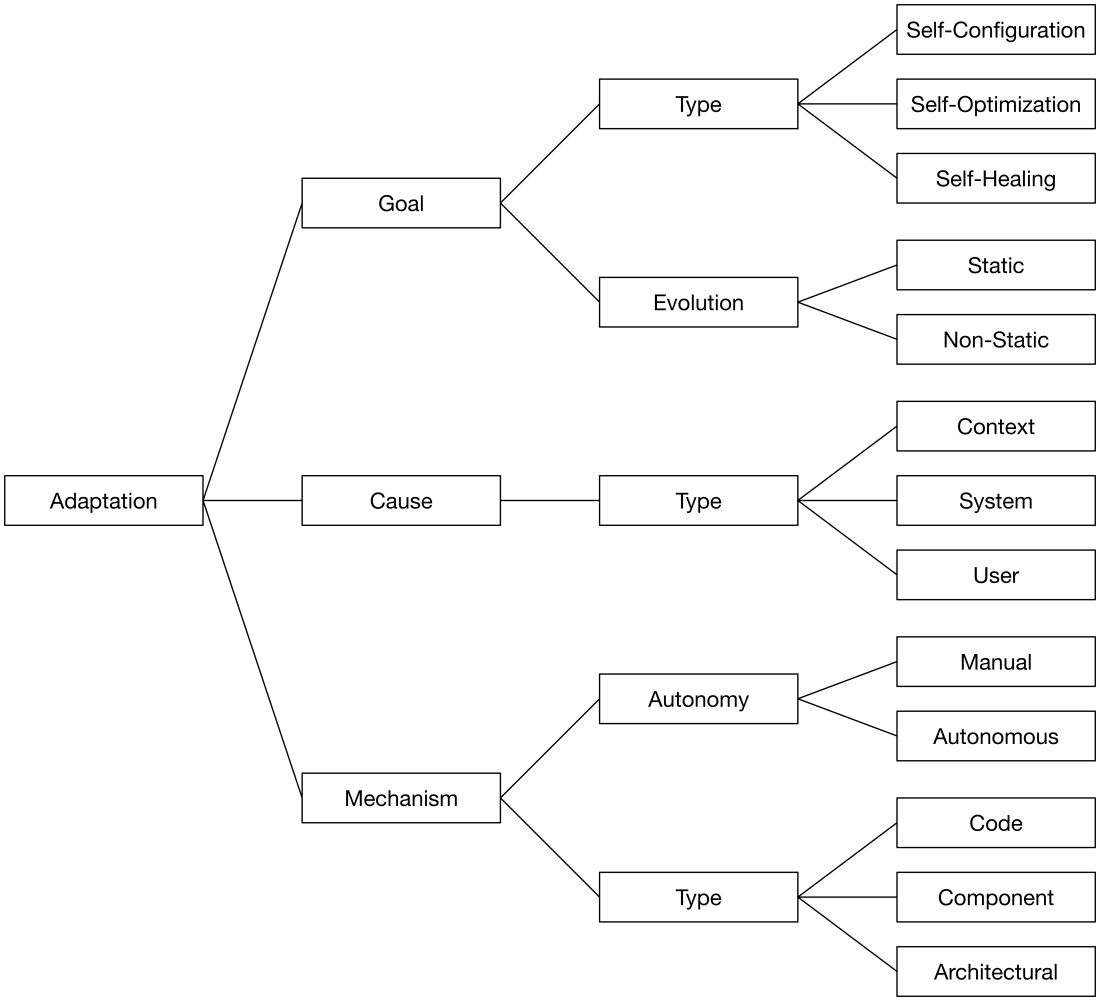

Figure A.1.: Adaptation Taxonomy [BBD16]

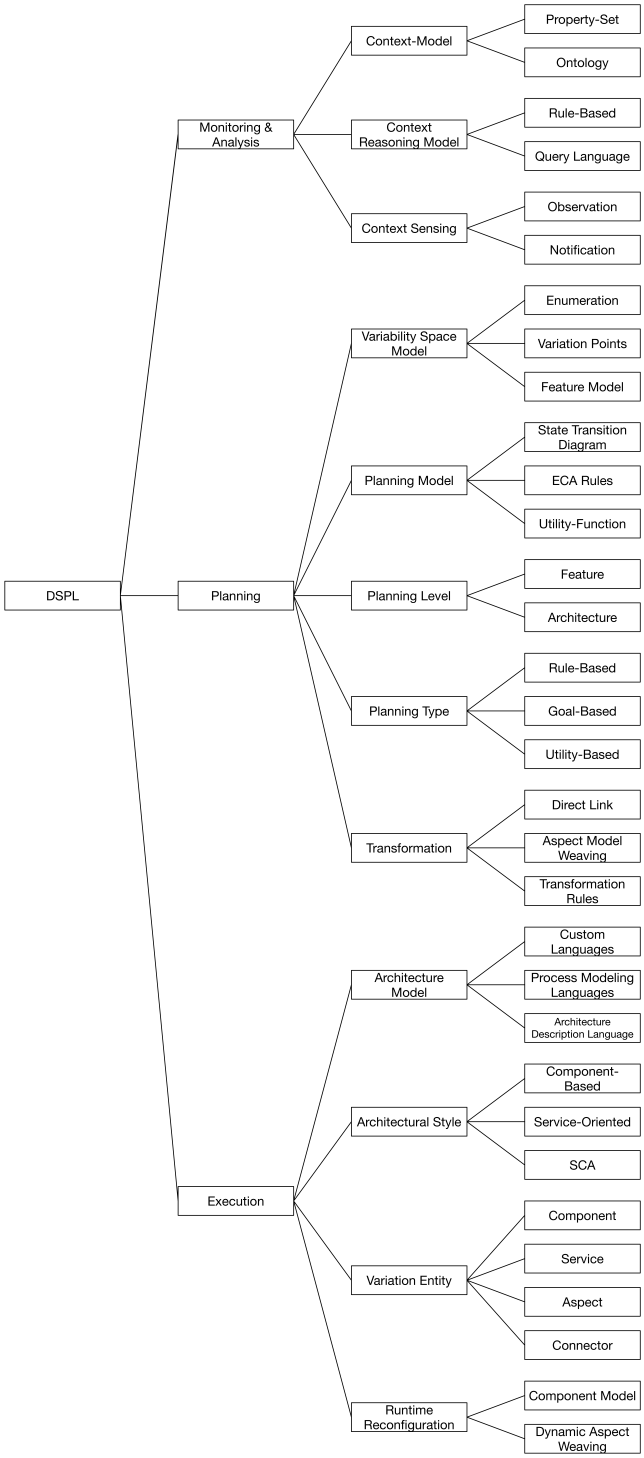

Figure A.2.: DSPL Taxonomy [BBD16]

Approach	Goal		Cause	Mechanism	
	Type	Evolution	Type	Autonomy	Type
Baresi et al. (S.-Or. DSPLs)	-	static	user	manual	code-level
Bencomo et al. (Genie)	self-configuring	static	context	autonomous	component
Cetina et al. (MoRE)	self-healing / self-configuring	static	context	autonomous	component
Floch et al. (MADAM)	self-optimizing	static	context / system	autonomous	component
Gomaa et al. (REPFLC)	-	static	-	autonomous	component
Morin et al. (DiVA)	-	non-static	-	autonomous	component
Parra et al. (CAPucine)	self-configuring	static	system	autonomous	component

Table A.1.: Characterization of the approaches using the adaptation taxonomy [BBD16]

Approach	Monitoring and Analysis			Planning				Execution			
	Context Model	Context Reasinong Model	Context Sensing	Variability Model	Planning Model	Planning Level	Transformation	Architecture Model	Architecture Style	Variation Entities	Middleware
Baresi et al. (Service-Oriented DSPLs)	N.A.	N.A.	N.A.	Feature Model	N.A.	Feature-Level	Direct Link	BPEL	Service-Oriented	Aspect	Dynamic Aspect Weaving
Bencomo et al. (Genie)	Property-Set	Rule-Based	N.S.	Enumeration / Variation Points	Transition Diagram	Feature-Level	Direct Link	OpenCOM DSL	Component-Based	Subsystem	OpenCOM
Cetina et al. (MoRE)	OWL	Query Language	N.A.	Feature Model	ECA Rules	Feature-Level	Direct Link	PervML	Service-Oriented	Service / Connector	OSGi Framework
Floch et al. (MADAM)	Property-Set	N.S.	MADAM Context Manager	Variation Points	Utility Function	Architecture	N.A.	N.S.	Component-Based	Subsystem	Configurable Product-bases
Gomaa et al. (REPFLC)	N.S.	N.S.	N.S.	N.A.	ECA Rules	Architecture	N.A.	N.A.	Component-Based	Component	N.S.
Morin et al. (DiVA)	Property-Set	Query Language	WildCAT	Feature Model	N.S.	Feature-Level	Aspect Model Weaving	Fractal / OpenCOM DSL	SCA / Component-Based	Component	Fractal / OpenCOM
Parra et al. (CAPucine)	Property-Set	Rule-Based	COSMOS	Feature Model	ECA Rules	Feature-Level	Aspect Model Weaving	Fractal	SCA	Component	Fractal

Table A.2.: Characterization of the approaches using the DSPL taxonomy of [BBD16]