Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master's Thesis

Dynamic Consistency Checking of Cloud Computing Patterns

Steven Großmann

Course of Study:	Software Engineering
Examiner:	Prof. Dr. Dr. h. c. Frank Leymann
Supervisor:	Dr. rer. nat. Uwe Breitenbücher
Commenced:	January 5, 2018
Completed:	July 5, 2018

Abstract

Cloud computing patterns can be used to build cloud applications based on proven and tested solutions. However, the correct implementation of these patterns is not always warranted during the whole life cycle of an application. Inconsistencies between implementation and cloud computing patterns can result in architectural drift and negatively impact the quality attributes of an application. Therefore, this thesis presents a method to check the consistency of cloud computing patterns at runtime. Nine cloud computing patterns are selected and analyzed in detail for structural and semantic constraints. A formal notation is developed to express the constraints and enable automatic consistency checking of the patterns. Furthermore, a framework is designed which enables monitoring of cloud applications and checking of constraints for cloud computing patterns at runtime. To show the feasibility of this method, the framework is implemented and successfully tested in a cloud computing environment, with a simulated workload on virtual machines, for the patterns *Static Workload*, *Continuously Changing Workload*, *Elasticity Manager*, and *Watchdog*.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

AWS Amazon Web Services. 26

BPMN Business Process Model and Notation. 37

CCPCC Cloud Computing Pattern Consistency Checking. 71

CEP Complex Event Processing. 26, 27

CPU Central Processing Unit. 39

DMMN Declarative Application Management Modeling and Notation. 31

DSL Domain Specific Language. 26

EBNF Extended Backus-Naur Form. 57

EDP Elemental Design Pattern. 22

EPL Event Processing Language. 27

FIFO First In - First Out. 72

GQM Goal-Question-Metric. 38

IaaS Infrastructure as a Service. 34

IoT Internet of Things. 19

IT Information Technology. 15

MDD Model Driven Development. 67

MTBF Mean Time Between Failures. 40

MTTR Mean Time To Recovery. 40

OCL Object Constraint Language. 23

ODOL Object-Oriented Design Ontology Layer. 23

OWL Web Ontology Language. 23

PaaS Platform as a Service. 34

RDF Resource Description Framework. 23

SOA Service-Oriented Architecture. 26

SQL Structured Query Language. 27

TOSCA Topology and Orchestration Specification for Cloud Applications. 31

UML Unified Modeling Language. 22

URI Uniform Resource Identifier. 23

XMI XML Metadata Interchange. 24

1 Introduction

The cloud computing paradigm has become an important part of contemporary Information Technology (IT) management in the industry. Outsourcing IT infrastructure in a cloud environment is a trend that concerns small and big companies [24]. The benefits of cloud computing are versatile, e.g. infrastructure can be dynamically provisioned and decommissioned within seconds and with minimal management effort. This increases flexibility and reduces costs, as the company only pays for what it actually uses. Though, developers of cloud applications have to deal with the dynamic, virtualized, distributed and multi-tenant nature of a cloud environment, which can result in issues [18]. Developing large software systems in a heterogeneous environment is complex and requires careful planning of the system.

Software architecture is essential to manage complex and large software systems [2]. However, it is difficult to create a suitable and flexible software architecture design on the first attempt. A good design is mostly the result of iterative attempts to implement it [27]. The usage of software architecture design patterns can help in this regard, as the patterns describe already tested and proven solutions to properly develop a software system. That means, design patterns are applicable for a certain or similar class of problems that lead to well-structured, maintainable and reusable software systems [58]. Software engineering has been strongly enhanced by design patterns, especially in the area of object-oriented programming [16]. During the years, several new patterns have been introduced for different domains like e.g. for cloud computing. Fehling et al. [24] express a cloud computing pattern as a well-defined document that describes a good solution to a cloud computing-related problem. Those patterns advise software architects of cloud applications on how to create proper architectures, define important design decisions, and cover limitations to consider [24].

However, the correct implementation of cloud computing patterns is not always warranted during the whole life cycle of an application. Maintenance tasks or the development of new features modify the initial system. Changes in the implementation may not always be updated in the documented architecture design, or documentation may accidentally or even consciously be ignored in order to quickly implement a work around solution. Therefore, systems can deviate from the original design specification which can have an impact on software quality aspects like, e.g. reliability, security, or maintainability [2]. The inconsistencies between implementation and architecture design are commonly referred to as architectural drift, or similar terms such as architectural erosion or architectural decay [13]. In order to detect inconsistencies, it is important to align architecture design with the actual implementation, which is referred to as software architecture compliance [13]. The alignment can be performed either dynamically at runtime, e.g. by monitoring the

application, or statically at design time, e.g. by source code analysis. Especially, continuous monitoring of a running application can help to keep the implementation faithful to the intended architecture and quickly detect inconsistencies [14].

The cloud computing patterns of Fehling et al. [24] are described in a well-defined format and written in natural language. This notation eases comprehension and enables uncomplicated communication among software architects. However, it is an informal notation which may result in imprecision and ambiguity during implementation or adherence checking. An informal pattern description does not allow for a fully automatic discovery process of design patterns [56]. As design patterns are often implemented as part of software architecture, it is important to automatically check the adherence of patterns during runtime, which implicitly requires a formal definition of those patterns.

1.1 Research Objectives and Contributions

Software architecture compliance regarding design patterns is a heavily investigated field of research, that has been studied since the mid 1990s and is still prevailing. While there are various approaches that allow consistency checking of object-oriented design patterns, especially in monolithic software systems, so far there is only little research in checking cloud computing patterns. Those patterns have a different nature compared to object-oriented design patterns and mostly deal with distributed applications in a heterogeneous environment. Hence, most of the existing approaches for object-oriented design patterns cannot be applied to a cloud computing environment.

The objectives of this Master's Thesis embrace the analysis of selected cloud computing patterns[1] for structural and semantic constraints. In this regard, a suitable notation must be documented which enables the formal description of structural and semantic constraints of selected cloud computing patterns. Furthermore, a framework must be designed and implemented which facilitates the consistency checking of cloud computing patterns at application runtime. The objectives have to be combined in a method to achieve dynamic consistency checking of cloud computing patterns.

The contributions of this Master's Thesis are:

1. A method for dynamic consistency checking of cloud computing patterns
2. Constraint definitions for selected cloud computing patterns of Fehling et al. [24]
3. A domain specific language to formally describe structural and semantic constraints for selected cloud computing patterns
4. An expendable framework to dynamically check the consistency of selected cloud computing patterns in cloud applications

[1] We focus on the cloud computing patterns of Fehling et al. [24] and the according website http://www.cloudcomputingpatterns.org [23]

1.2 Thesis Structure

The following chapter discusses fundamental concepts and presents related approaches and methods. Chapter 3 explains and reasons about the method that is used to achieve dynamic consistency checking of cloud computing patterns. In Chapter 4, the patterns of Fehling et al. [24] are analyzed for suitability regarding the developed method and for structural and semantical constraints. The development of the constraint language is presented in Chapter 5, which includes a meta-model, syntax, and semantics definition. Chapter 6 describes the design and implementation of the consistency checking framework. The method is tested by the means of example constraints for selected patterns in Chapter 7. Eventually, Chapter 8 summarizes the findings of this thesis and provides an outlook for future work.

2 Background and Related Work

This chapter provides the background information that is required to understand the research area and the methods that are used in the approach of this thesis. Additionally, existing approaches of related work are discussed by explaining their intent, providing a short description, and describing advantages and drawbacks. The sections address design patterns, software architecture compliance checking, and application monitoring technologies.

2.1 Design Patterns

Alexander et al. [4] first introduced patterns as nuggets of advice that describe solutions to recurring problems in a specific domain. Even though Alexander was speaking of patterns for building structures and cities, patterns can also be applied as design in the area of software development, as done by Gamma et al. [26][27]. Software architecture design patterns[1] provide a means to apply well-tested and successful software solutions, to a problem that frequently occurs in a certain context. Hence, especially novice software developers can use design patterns as a guide to create robust and flexible software architectures, as beginners mostly cannot resort to a lot of experience in designing software systems.

Collections of patterns are often connected and interrelated, forming a pattern language [24], which is formally defined by Falkenthal et al. [21]. Informally, a pattern language is described as a web of related patterns that are guiding the reader through the decision-making process, aiming at solving a limitless number of problems within a certain problem space [32]. Furthermore, patterns can be expressed on different levels of abstraction, whereas a high degree of abstraction makes patterns very reusable [22]. However, concrete patterns are easier to implement for a certain use case and, thus, patterns can be refined by linking them even across pattern languages, to reduce the abstraction gap between abstract knowledge and concrete actions [22].

2.1.1 Domains

The most popular and widely discussed design patterns in software engineering were presented by Gamma et al. [26] for the domain of object-oriented programming. Yet,

[1] In the following this term will be shortly referred to as design pattern

there are various domains in software engineering that employ design patterns, e.g. cloud computing patterns [1, 12, 24, 29, 63], Internet of Things (IoT) patterns [50, 51], enterprise integration patterns [32], and further more. Especially cloud computing patterns have gained popularity during the last years, because the paradigm of cloud computing is utilized more often in the industry. Thereby, multiple different collections of cloud computing patterns have been published, both by cloud vendors and by academical researchers [16]. According to the classification of Di Martino et al. [16], cloud computing patterns defined by cloud providers will be referred to as proprietary patterns [1, 29, 63]. On the other hand, patterns as a consequence of academical research are accordingly defined as agnostic patterns [12, 23, 24].

2.1.2 Format

The documentation of a pattern is mostly done in a specific format, e.g. written in prose text with additional graphical notations. Even though there is no universal pattern form [32], most documentations share a similar format. That means, they describe the pattern structure, static and dynamic aspects, and various ways to implement it [52]. According to Gamma et al. [27] a pattern commonly consists of four fundamental elements, i.e. the pattern name, a section describing the problem statement, a section describing the solution approach, and a section that discusses the consequences of applying the pattern. Hence, other design pattern books of e.g. Hohpe et al. [32] and Fehling et al. [24] describe patterns in a similar fashion. In this work we will mainly focus on the cloud computing patterns of Fehling et al. [23, 24], thus we will shortly describe the pattern elements and their meanings.

The *Pattern Name* identifies a design pattern and can serve as a reference within the pattern language. Directly following the pattern name, there is an *Intent* section that provides a quick overview of the purpose and goal of the pattern. A *Driving Question* allows readers to decide whether the pattern is suitable for their problem or not. An *Icon* serves as a graphical representation of the pattern, that can be used in an architectural diagram or in sketches. The *Context* section describes the environment, forces, and sometimes naive solutions that can be unsuccessful or suboptimal. The *Solution* section briefly describes how the pattern solves the problem, raised by the driving question and is important to identify structural and behavioral aspects. A more detailed explanation of the solution is elaborated in the *Result* section. Additionally, the application behavior is explained, which makes the result section an important element for an in-depth pattern analysis. A pattern may be applied in slightly different forms, that are explained in the *Variations* section. Interrelations between patterns are mentioned in the *Related Patterns* section. The *Known Uses* section depicts existing applications that implement cloud computing patterns. This section is essential for implementing test scenarios for the validation of the developed method.

2.1.3 Formal Notations

Design patterns have a dual nature as they can be expressed both formally and by informal descriptions [56]. The description of design patterns in catalogs, books, journals and other publications is mostly expressed in non-formal representations [16, 36]. An informal representation of design patterns, e.g. in prose text, is useful for communication among developers, but can be ambiguous and inadequate for automated processing [16, 25]. To achieve greater precision and facilitate automated compliance checking of design patterns it is necessary to utilize formal notations, that describe pattern aspects visually, textually, mathematically, or by a combination thereof. There are two main aspects that have to be formally described by such a notation, i.e. the static aspects and the dynamic aspects [3, 31]. Static aspects of a pattern define the properties that do not change the state and are defined at design time, e.g. system structure, or component properties. On the other hand, dynamic aspects define the behavior of a system, i.e. how the state and attributes of a system change during runtime, e.g. method calls, or scaling of components. There are pros and cons for the different types of notations, which will be addressed in the following subsections.

Mathematical and Textual Notations

Design patterns can describe an abstract structure for programming a solution, but ultimately they are expressed as source code that can be reduced to a mathematically formal expression [56]. Hence, the precision and unambiguity of mathematical formulas can serve as a way to formally describe design patterns. Furthermore, there are (non-prose) textual approaches that define syntax, semantics, and inference rules which are the key to an unambiguous formal language [20]. The following approaches and methods use a mathematical notation and textual notation, respectively, to exactly express design patterns.

Amnon H. Eden describes an approach to formally define object-oriented design patterns by using a formal language called LePUS [20]. His approach is driven by the motivation to overcome the limitations of visual diagrams that are annotated with informal comments. The definition of LePUS is based on the desirable properties that an architectural specification should posses, as described by Perry and Wolf [49] and Eden [20]: The language should (1: Generality) allow the expression of detailed constraints, (2: Abstraction) allow a concise expression of high-level constructs, (3: Elementary building blocks) have a small set of design elements that allow the expression of complex composite structures, (4: Formal) have well defined, unambiguous semantics and should allow adequate reasoning, (5: Concise) have commonly occurring motifs expressed short, (6: Compact) and follow the principle, fewer words make a more powerful language. LePUS meets all the mentioned properties and is specifically tailored for object-oriented architectures. The language facilitates tool support by the translation of LePUS formulas into the PROLOG language. France et al. [25] and Vu et al. [60] argue, that LePUS is precise and unambiguous, but may be difficult to understand because of the mathematical syntax. However, because

of the underlying formalisms of LePUS, the formulas can be expressed in a semantically equivalent, graphical notation.

Smith et al. [56] developed an approach to mathematically express design patterns of Gamma et al. [27]. The intention of Smith et al. is not to completely formalize a design pattern description, but rather extend design patterns by an as formal as possible notation. They found out that design patterns can be expressed by only a few Elemental Design Patterns (EDPs), that are usually not thought of when designing or developing a system. With the usage of ς calculus, an object-oriented analogue to λ calculus, they managed to express parts of those EDPs. However, with ς calculus it is not possible to define relationships between constructs, which is why they extended it with the use of reliance operators, resulting in the ρ calculus notation [56]. Smith et al. created a formal notation while maintaining the flexibility of pattern implementations. Their approach facilitates tool support as they showed by implementing a System for Pattern Query and Recognition (SPQR), to improve code comprehension. Although, using a mathematical notation requires sophisticated mathematical skills [25] and, hence, their notation lacks usability for software architects and developers.

Taibi et al. [57] developed a mathematical notation that formally describes all aspects of object-oriented design patterns. The motivation for their approach is driven by the drawback of existing approaches, that have the tendency to either focus on the structural or behavioral aspects of design patterns [57]. Their notation, called Balanced Pattern Specification Language (BPSL), combines two subsets of logic, i.e. First Order Logic (FOL) and Temporal Logic of Actions (TLA). Taibi et al. provide a formal language that fully combines two complementary aspects, meaning structure and behavior of patterns. Hence, BPSL is a complete solution for developers and architects that helps to understand design patterns in detail, to decide when to use it, and how to use it [57]. However, BPSL specifications appear to be verbose and contain mathematical symbols, which can result in reduced readability, and decrease communication among domain experts [36].

The approach of Heuzeroth et al. [31] uses a textual notation that is based on PROLOG to express object-oriented design patterns. They developed two notations to address both, the static aspects in the form of structural connections and the dynamic aspects as protocol of actions. The combination of both aspects is used to detect design patterns in legacy code and, thus, elaborate software comprehension. The first notation SanD-Prolog is basically a collection of PROLOG predicates that present relations and state transitions of a pattern. With SanD-Prolog it is possible to express the structure and the behavior of design patterns, however, the specifications can be too long and complicated [31]. Consequently, they developed a more high-level and intuitive specification language called SanD. The syntax is kept similar to existing object-oriented languages and, hence, is more familiar to developers. SanD is a concise and intuitive language that can express the structure and behavior of design patterns. However, the language is not as powerful as the first notation SanD-Prolog, as constraints cannot express to omit constructs like, e.g. a class must not contain a certain method.

Visual Notations

Visual notations are often used in the area of software architecture design, as e.g. by using Unified Modeling Language (UML). Diagrams provide a quick overview but sometimes may lack precision and, thus, are additionally annotated with informal textual annotations [36]. The following approaches use visual notations similar to, or even based on UML.

Lauder et al. [36] developed a visual modeling notation to formally describe the object-oriented design patterns of Gamma et al. [26]. Their approach is motivated by solving the demerit of superfluous descriptions of specific instances in design patterns and informal diagrammatic notations, supplemented with natural language. They propose a separation of pattern descriptions into three models, i.e. a role model, a type model, and a class model. The role model first describes the pattern's essential spirit, which tackles the demerit of superfluousness in pattern descriptions, then the type model adds domain specific constraints, and lastly, the class model forms a concrete deployment to facilitate tool support for design patterns. This separation allows the pattern designer to operate at a higher level of abstraction without ambiguity. Their visual diagrams are more precise than the UML diagram descriptions as used by Gamma et al. [26]. For example, the UML representation of the Abstract Factory pattern misinterprets quantities and shows premature commitment to class names. If the pattern user does not interpret the UML diagram correctly, it might lead to a misuse of the pattern. Using the approach of Lauder et al., a misinterpretation is prevented by the essence of the three-model layering, which achieves abstract-yet-precise expressiveness via a set-oriented representation of state, operations, and instances [36]. However, France et al. [25] argue that the notation of Lauder et al. is not integrated into UML and, thus, tool support is poor which decreases feasibility of the approach. Furthermore, meta-modeling layers based on UML [46] diagram representations in combination with Object Constraint Language (OCL) [45] constraints can result in more precision and remove ambiguities, too.

France et al. [25] present a visual approach to formally describe object oriented design patterns expressed in a syntactic variant of UML. The goal of their work is to provide a pattern specification technique that is easily accessible for UML modelers and enables the use of tool support for creating precise pattern specifications. A pattern is defined by a structural specification, specifying the class diagram view of a pattern, and an interaction specification that describes the pattern behavior. Furthermore, they enrich the structural specification by well-formedness rules expressed in OCL constraints. The benefit of the approach of France et al. is that tool support can easily be built because the notation is UML-based. However, especially the expression of pattern behavior and recursive tasks with their interaction notation is not fully satisfactory according to their conducted experiments. They investigated the usage of UML 2.0 sequence diagram notation instead, which offers a richer set of constructs [25]. This questions the relevance of their notation, when instead a pure UML notation offers even more constructs to precisely define design patterns.

UML diagrams are seen as the de facto standard in object-oriented modeling [25] and, thus, are popular when describing design patterns [17]. Although, UML diagrams do not support variables that are required to formally describe patterns [17, 19]. Dietrich et al.

[17] developed an approach that is based on the Web Ontology Language (OWL) [47], which they argue is more formal and precise compared to UML. The idea is to represent software artifacts identified by Uniform Resource Identifiers (URIs) and to develop a set of relationships between constraints which are used to describe design patterns. By using formal languages like Resource Description Framework (RDF) and OWL, these patterns can be expressed with formal semantics and well-defined syntax, which eliminates ambiguities and facilitates tool support. Furthermore, this approach yields an extensible description of design patterns that is independent to programming languages. The drawback of the described approach is, that it is solely based on the Object-Oriented Design Ontology Layer (ODOL), which focuses on the definition of structural aspects of design patterns and, hence, neglects dynamic behavior.

Di Martino et al. [16] also developed an approach to formally define design patterns utilizing OWL and a machine-readable representation of UML diagrams coded in XML Metadata Interchange (XMI). The usage of OWL is a popular method to formally define design patterns, as many more similar approaches based on OWL exist. Although, Di Martino et al. argue that previous approaches based on OWL, e.g. like [17, 35, 60], provide incomplete pattern descriptions because they neglect dynamic behavior. Furthermore, ODOL is mainly developed for object-oriented design patterns, which does not allow to properly define OWL descriptions for patterns of other domains, as e.g. for cloud computing patterns [24]. Hence, they extended the ODOL structure to allow the definition of various pattern domains and combined pattern descriptions with OWL-S to support the expression of behavioral aspects. OWL-S is an OWL based language, that is flexible, extensible, and can be used to describe workflows, which is why they use it to express the behavior of patterns. The advantages of their approach is the possibility to define structural and behavioral aspects of patterns, not only restricted to the classical object-oriented patterns as e.g. [27]. Furthermore, the diagrams can be expressed in a machine-readable form which facilitates tool support, as they proofed with a prototype that detects design patterns in applications. Even though they claim that their approach is applicable to more recent patterns, as e.g. cloud computing patterns [24], they only tested it on a few object-oriented patterns.

2.2 Software Architecture Compliance Checking

Design patterns can be seen as micro architectures that can be combined to compose a component or even define the whole architecture of a system [57]. To align software architecture design with the actual implementation and automatically check compliance, it is necessary to have both, design patterns and source code, in a machine-processable notation. By using a formal notation of design patterns, it is possible to develop tools that support machine readable forms of the patterns, as explained for different approaches in Section 2.1.3. The actual implementation is already in a machine-readable and formal notation, i.e. source code, and can be checked directly.

There are two main methods on how to check design patterns and software architecture, respectively, i.e. either by design time (static) verification or by runtime (dynamic) analysis

[13]. As this thesis aims on runtime checking I will focus in the related work on approaches and methods that check design patterns and architecture using runtime analysis.

Runtime checking allows to detect the actual application behavior, e.g. a connection between components might be given at design time, however, at runtime the connection might be omitted. There are two ways for runtime checking [41], i.e. by using offline analysis, where all important actions are recorded at runtime and analyzed after termination, for example in a log file. The other variant is online analysis, where the application is both, checked and analyzed during system execution. The following approaches give an overview of various runtime checking approaches that aim on detecting design patterns in applications and verify the consistency to the design.

2.2.1 Offline Analysis

Ackermann et al. [2] built their approach based on the reflexion-models of [43] and adopted them to support compliance checking of behaviors. The approach aims on checking the architecture of distributed systems at runtime by recording message communication between system components. After execution, the messages are mapped according to their type and context to the high-level model, that is described as UML sequence diagram. The resulting reflexion-model is computed by matching algorithms and shows the deviation between design and implementation. The drawback of their approach is that it is restricted to a limited set of sequencing constructs [2] and, thus, is not feasible to be applied on large systems.

Wendehals [61] developed an offline approach for design pattern recognition in monolithic object-oriented applications. The approach aims on helping reverse engineers to recover application architecture that is not documented properly. First, they use static source code analysis to extract pattern instance candidates that serve as the base for further analysis. The dynamic analysis utilizes debugging programs to capture method call traces of the executed application at runtime and then creates an attributed call graph. Next, they use sequence diagrams of design pattern specifications, in the form of graph rewrite rules, and apply them to the recorded call graph. In the end, the result of the analyzed call graph is compared to the results of the previous static source code analysis to decide whether a certain pattern instance candidate actually adheres to the design pattern specification. The combination of static and dynamic analysis reduces the execution time of the process and achieves greater precision in pattern recognition compared to only static analysis [61]. However, this approach is limited to the use of absolute quantities of objects in the sequence diagrams and, thus, increases the probability that pattern instance candidates are incorrectly labeled as false positive [15]. With the introduction of set objects, presented by Detten et al. [15], this limitation is bypassed and allows to detect arbitrarily large sets of objects in design patterns.

2.2.2 Online Analysis

The advantage of online analysis approaches, compared to offline analysis, is that compliance violations are reported in near real time, without requiring human intervention after program execution. Another approach by Wendehals and Orso [62] employs a method that is related to online monitoring. Again, static analysis serves as the foundation for the dynamic analysis to identify relevant pattern instance candidates and instrumenting relevant method calls in the analyzed program. Furthermore, for the dynamic analysis, the behavioral aspects of design patterns are translated into a finite automaton. The program execution monitors the instrumented method calls at runtime and matches them against the automatons. Each method call trace that matches a design pattern behavior definition increases the confidence, in that the pattern instance candidate actually adheres to the design. A rejection state is used for traces that are not in order with the automaton and with the pattern behavior, respectively. Multiple executions of an automaton are prevented by the introduction of tokens, similar to tokens in petri nets. Although, the approach might fail in case the analyzed patterns comprise too simple dynamic behavior, because this would produce too many pattern matches and results in a high rate of false positives [62].

Similar to design patterns, process models can serve as the foundation for checking the architecture compliance of an application. Mulo et al. [41] present an online monitoring approach, where they monitor the execution of applications implementing the process-driven Service-Oriented Architecture (SOA) paradigm. They developed a Domain Specific Language (DSL) to translate business processes into an event-based sequence. Whenever a service in the workflow is invoked, a service invocation event is emitted to monitor the process sequence. Complex Event Processing (CEP) is used to handle all events and search with event detection patterns, that are mapped from the DSL, whether the execution sequence is compliant to the specified process. The approach allows to detect anomalies in the implementation in near real time and because of the DSL, the approach is agnostic to specific technologies. However, a drawback is that the emitting of events has to be included into the source code of the services, which is cumbersome and sometimes not even possible.

2.3 Cloud Application Monitoring

Monitoring plays a significant role for checking cloud computing applications at runtime, as the components of cloud applications are constantly subject to change, considering scaling, defects, or discovering usage patterns [55]. However, the heterogeneity and complexity of cloud computing makes it difficult to keep track of application runtime behavior [55]. Approaches for architecture compliance checking, as shown in Section 2.2, mostly consider techniques to monitor method invocations in the source code by e.g. utilizing debugging functionality as used in [61, 62]. This is not feasibly applicable in the domain of cloud computing, as applications can consist of multiple and distributed services that are implemented with different languages and technologies.

2.3.1 Cloud Provider Monitoring Services

Well-known cloud providers in the industry provide own services to monitor components in the respective cloud environment [55]. E.g. Amazon CloudWatch[2] enables to monitor Amazon Web Services (AWS) cloud resources for extracting metrics, creating protocols, and defining alarms based on changes of AWS resources. Similar monitoring services for other cloud platforms are for example provided by the products Microsoft Azure Monitor[3], Google Stackdriver Monitoring[4], or IBM Cloud Monitoring[5]. They all have in common that the monitoring is restricted to monitor their own cloud platform, or a few specific other platforms (e.g. Google Stackdriver can monitor the Google Cloud Platform and AWS). Furthermore, free usage is restricted to imprecise monitoring, as e.g. for AWS CloudWatch where virtual machine usage is only updated every five minutes.

2.3.2 Cloud Provider Agnostic Monitoring

Implementing a monitoring facility that is agnostic to cloud providers can be a challenging task, because of the large number of services and their heterogeneity [55]. Shao et al. [55] solved this issue by developing a runtime model for cloud computing and an according framework that monitors various types of cloud components in a cloud. By deploying monitoring agents on critical cloud resources, e.g. on virtual machines, they enabled to extract runtime information and send it to the monitoring framework.

Monitoring is also widely used in the area of compliance checking for SOA, which shares similar properties to the domain of cloud computing, e.g. loose coupling, dynamism, and heterogeneity [55]. Multiple approaches on compliance checking for SOA are developed on the basis of processing events, e.g. [41, 42, 53]. Hence, a monitoring technology that is applicable for heterogeneous and distributed applications is Complex Event Processing (CEP), which was first introduced by David Luckham in [38][53]. The term defines a set of technologies to process complex series of interrelated events, which allows monitoring and control of enterprise applications in real time [41, 53]. Therefore, this technique is usually applied in areas that require low latency times for decision-making processes, as for example in financial market analysis, trading, security, fraud detection, and compliance checking [53]. In the domain of CEP, there are two kinds of events, i.e. low-level events and complex, or high-level events, respectively [41, 42]. Low-level evens are not an abstraction of other events and do not have semantic significance on their own, whereas complex events are an aggregated abstraction of a number of other low-level or high-level events [41, 42]. The usage of Event Processing Languages (EPLs) allows to query event streams and configure event processing engines to aggregate events into complex events. Those languages are often expressed in the form of extended Structured Query Language

[2] https://aws.amazon.com/de/cloudwatch/
[3] https://docs.microsoft.com/de-de/azure/monitoring-and-diagnostics/monitoring-overview-azure-monitor
[4] https://cloud.google.com/monitoring/
[5] https://console.bluemix.net/catalog/services/monitoring

(SQL) statements, as e.g. for Oracle Continuous Query Language[6], Apache Flink SQL[7], or Esper EPL[8]. That means, the languages mostly use ANSI SQL statements, like e.g. select-clauses, from-clauses, etc., and extend them by the definition of event processing relevant statements, as e.g. the definition of time windows and event sequences.

2.4 Conclusion

Various attempts were developed to formalize patterns and their aspects by utilizing different notations. Mathematical notations are considered to be the most formal way to express design patterns and their aspects, as they are unambiguous and precise. However, mathematical notations are also considered to be very complex to understand and are not applicable in the daily business of software design and development. In contrast, visual notations are often used in software architecture design and are more easy to understand. The drawback of visual notations is that they can be ambiguous and, then, must be annotated with textual explanations. Furthermore, when using diagrams, the representation of patterns can become big and confusing. A compromise between mathematical and visual notations are textual notations, which are usually inspired by existing languages, as e.g. general purpose programming languages. Textual notations enable formal expressions of design patterns and aspects in a machine-readable and clear arranged fashion.

When having patterns or according aspects, respectively, in a formal notation, it is possible to check the consistency between application design and implementation. There are different approaches for runtime checking of applications, which either use offline or online analysis. Especially online approaches are valuable to detect inconsistencies between design and implementation in (near) real-time. To enable runtime checking, it is important to use a suitable monitoring method, which is available for all popular cloud providers. Although, for being able to monitor applications without being restricted to a certain provider, it is necessary to implement own monitoring methods. Regarding cloud applications, CEP is a suitable technology to keep track of runtime changes, as also applied for compliance checking in related domains.

[6] https://docs.oracle.com/cd/E16764_01/doc.1111/e12048/intro.htm
[7] https://ci.apache.org/projects/flink/flink-docs-release-1.5/dev/table/sql.html
[8] http://esper.espertech.com/release-5.2.0/esper-reference/html/epl_clauses.html

3 A Method for Dynamic Consistency Checking

This chapter presents a method to achieve dynamic consistency checking of cloud computing patterns. Furthermore, it is reasoned about the decision for the development of a constraint language, explained which technologies have to be considered for the framework implementation, and the domain for the method is presented.

3.1 Overview

To dynamically check the consistency of cloud computing patterns, a method is defined which uses formal constraints for a monitoring framework that checks the consistency between design and implementation at runtime. Initially, it is necessary to analyze the cloud computing patterns of Fehling et al. [24] for suitability. Detailed analysis is restricted to selected patterns, because there are over 70 patterns in [24] and not all of them are suitable for dynamic consistency checking. An overview of the method is depicted in Figure 3.1.

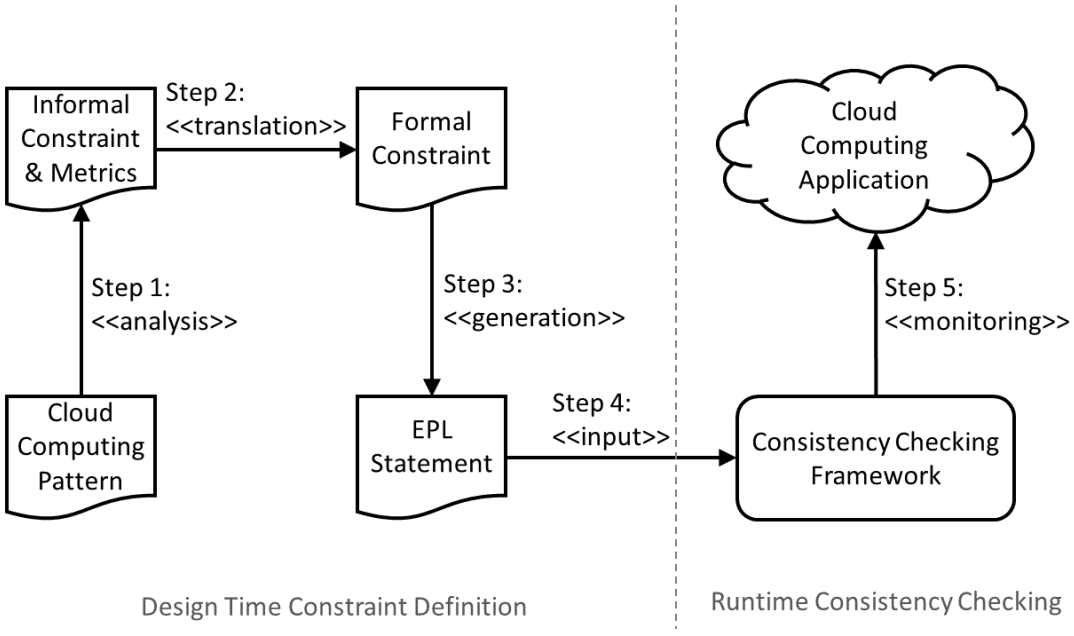

Figure 3.1: Overview of the Method

The method is divided into two phases, i.e., one, the Design Time Constraint Definition phase and, two, the Runtime Consistency Checking phase. The first step at design time addresses the analysis of selected cloud computing patterns, to extract constraints that describe structure and semantics, which is conducted in Chapter 4. The result of the analysis are constraints described in informal prose text. Those constraints have to be formalized in order to remove ambiguities and increase precision, which is achieved in Step 2 of the method by translating the informal constraints into a formal constraint language, that is specified in Chapter 5. The constraints, then, are machine processable and can be generated as EPL statements in method Step 3, to enable tool support for application checking at runtime. In Step 4, the EPL statements are used as input for the consistency checking framework, which is described in Chapter 6, to setup an event processing engine to query runtime information. As soon as the input is integrated in the framework, the second phase of the method is activated, which requires to start the monitoring process in Step 5. The consistency checking framework monitors a cloud application at runtime and creates events, out of the monitored information, which are injected into the event processing engine of the framework. This allows verifying, whether the implementation is consistent to the considered cloud computing patterns or not.

3.2 Constraint Language

There are various existing formal notations for design patterns and their aspects, as presented in Section 2.1.3. Although, most of the approaches and methods are developed for the domain of object-oriented design patterns, which differs from the domain of cloud computing patterns. Monolithic applications, as mostly considered in Section 2.1.3, are not designed to be executed in a cloud environment [24] and do not suit the domain of this method. Other notations, as e.g. specified by Di Martino et al. [16], can be applied for cloud computing patterns, although they are not designed for dynamic consistency checking methods. Thus, there is no existing formal notation that is suitable for the goal of this thesis and, consequently, it is necessary to develop a domain specific language.

3.2.1 Representation Format

The notation is developed to represent structural and semantic constraints for the selected cloud computing patterns. I decided for a textual notation, as mathematical notations are not feasible to be used by software architects and developers in daily business tasks. Conversely, a visual notation might not always be fully satisfactory when presenting multiple aspects of patterns, as mentioned in the related work [25]. According to the findings of Section 2.1.3, I developed a textual notation that is inspired by the syntax of OCL. A similar syntax to OCL helps to provide a short learning curve and increases the acceptance among domain experts, as OCL is widely known in the area of constraints for system design, considering the popularity of UML.

3.2.2 Dissociation from Event Processing Languages

As described in Section 3.1, the language is translated into EPL statements, which technically could also be used to formally present the constraints of cloud computing patterns. However, there are reasons to not use the EPL for pattern definition, as this would be a misuse of the domain which is addressed by the EPL. Vice versa, domain experts of cloud computing patterns would have to rethink the domain of the EPL, which is not beneficial in the constraint definition process. The notation of cloud computing pattern constraints would depend on a specific technology, which is equivalent to defining pattern constraints using a general purpose programming language as e.g. *Java*. When using an independent DSL, it is possible to switch the underlying technology in the consistency checking framework, which is also mentioned as a reason in related work, as e.g. explained by Mulo et al. [41].

3.3 Consistency Checking Framework

The framework must be capable of checking consistency at runtime, which is why properties of the monitored application have to be extracted in near real time. It is possible to use cloud provider specific monitoring techniques for this purpose, however, this would make the framework dependent on certain cloud platforms. Therefore, it is not optimal to rely on proprietary monitors, instead, it is more beneficial to apply methods as explained in the approach of Shao et al. [55]. By implementing monitoring programs for different types of cloud components, which are deployed on significant cloud resources as for example on virtual machines, it is possible to extract runtime information independent of the cloud platform. The information then has to be integrated and analyzed for certain characteristics that allow checking for structure and semantics. CEP is a suitable technology for this problem, as it allows to query events based on EPL statements in real time. Accordingly, the framework utilizes an event processing engine that is configured by constraints in the form of EPL statements and enriched with runtime information about the cloud application in the form of events.

3.4 Domain

The constraint language and monitoring framework defined in this method are specifically designed for the domain of cloud computing applications. Consequently, the constraint language is a DSL, for which it is necessary to specify a domain-model that addresses the environment and corresponding components [59]. Cloud computing environments provide a variety of IT-resources that are interconnected with each other. Those IT-resources can be of different types, e.g. infrastructure, middleware platforms, software, etc., and can be distributed among multiple logical and physical components, forming a distributed application [24]. The management and description of large cloud applications can be

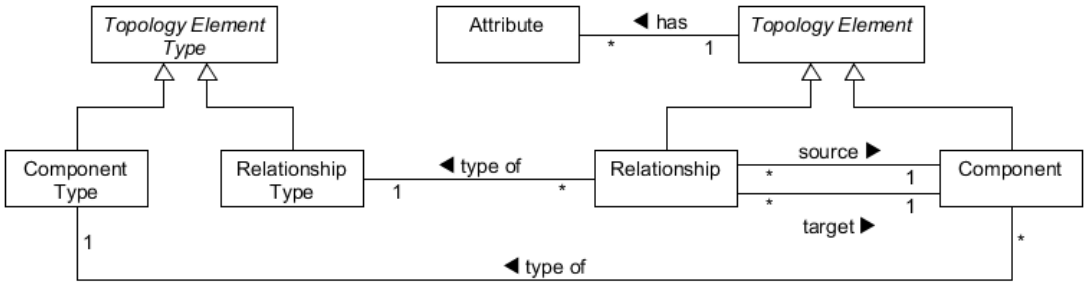

Figure 3.2: Domain-Model

a challenging task, thus, standards have been developed to clearly arrange and represent cloud applications. Topology and Orchestration Specification for Cloud Applications (TOSCA) [44] is an OASIS standard that allows the description of cloud applications and processes to manage them. Consequently, TOSCA would be a suitable candidate to describe the domain that is addressed by the constraint language for this method. However, the definition of constraints for cloud computing patterns should be agnostic to specific topology definition languages. The runtime cloud model of Shao et al. [55] is not suitable, too, as it defines too concrete types of components, which makes the model inflexible. Therefore, the domain-model is abstracted from a TOSCA related, but more adaptable language, i.e. Declarative Application Management Modeling and Notation (DMMN), which is introduced by Breitenbücher [7] and further refined by Saatkamp et al. [54].

Figure 3.2 shows the domain-model in UML class diagram notation and represents application topologies as a directed, weighted and not necessarily connected graph. A topology consists of *Components* and *Relationships*, which are *Topology Elements* that can hold multiple *Attributes*. IT-resources of a cloud application are represented as a *Component* that belongs to a certain type, which is expressed as *Component Type*. A *Relationship* is used as a connection between two components, i.e. a directed relationship between a source *Component* and a target *Component*. Each *Relationship* has a certain *Relationship Type* which describes semantics. Both, *Component Type* and *Relationship Type* elements are defined by the supertype *Topology Element Type*.

4 Analysis of Cloud Computing Patterns

This chapter describes the analysis of cloud computing patterns of Fehling et al. [24]. Before conducting a detailed analysis, suitable patterns are selected in Section 4.1. These selected patterns are then analyzed for structural and semantic constraints by defining metrics and constraint statements. The detailed pattern analysis in Section 4.2 and Section 4.3 corresponds to the first step of the method defined in Chapter 3.

4.1 Pattern Selection

The cloud computing patterns of Fehling et al. [24] are very diverse. They range from abstract descriptions of cloud environments to detailed specifications of processes, that combine cloud application components to solve a certain problem. Not all cloud computing patterns hold detailed descriptions of measurable properties, which are required to express a pattern by formal constraints. Some patterns do not exhibit variable runtime behavior, which makes dynamic consistency checking superfluous. In the following, the patterns defined in [24] are examined, i.e. whether a pattern is suitable for a formal description using structural and semantic constraints, that can be checked dynamically at runtime. The analysis is proceeded by examining all elements of cloud computing patterns (cf. Section 2.1.2 on page 20) and check whether the pattern is suitable according to the following selection criteria: The pattern must ...

- describe clear conditions and constraints under which the pattern is valid
- define measurable metrics
- hold variable runtime behavior
- be checked at runtime to verify consistency

Furthermore, it is reviewed whether a pattern can be expressed in a meaningful constraint and if it is feasible to check the pattern at runtime by a general consistency checking framework. The following analysis is structured according to the five categories of cloud computing patterns defined in [24], i.e. *Cloud Computing Fundamentals*, *Cloud Offering Patterns*, *Cloud Application Architecture Patterns*, *Cloud Application Management Patterns*, and *Composite Patterns*. In the last subsection (Section 4.1.6), all selected patterns are summarized in a table format.

4.1.1 Cloud Computing Fundamentals

The patterns of this category describe application workload, service models, and deployment types according to the NIST cloud definition [40]. These patterns differ from patterns of other categories, as they are not implemented by developers. Instead, they characterize the context in which other patterns are applicable and describe how cloud providers offer IT resources to customers. Because of the abstract nature and description of *Cloud Service Models* and *Cloud Deployment Models*, patterns of those subcategories are not considered to be suitable according to the defined selection criteria. A consistency check at runtime would not be meaningful, as cloud service models and cloud deployment models do not exhibit variable runtime properties. In this context it is more reasonable to use an approach as presented by Képes et al. [34] and Breitenbücher et al. [8, 9], in which policies are used to express non-functional properties in deployment models. Using the policies approach, patterns like e.g. *Private Cloud* can be expressed and checked in a meaningful way.

Application Workload patterns describe more specific metrics, i.e. workload experienced by applications. Application workload is a variable property that can change over time and is measurable. From the view of an IT architect, it may be of importance to know if the application experiences expected workload. Therefore, the patterns *Static Workload*, *Periodic Workload*, and *Continuously Changing Workload* are considered suitable. *Once-in-a-lifetime Workload* and *Unpredictable Workload* are measurable patterns, but they do not exhibit clear constraints that can be expressed substantial. Considering the viewpoint of an IT architect, I do not find the latter two patterns mentioned to be suitable.

4.1.2 Cloud Offering Patterns

Cloud Offering patterns describe cloud environment properties and offerings provided by cloud vendors, i.e. processing of workload, data storage, and communication. The offering patterns give advise on when to choose a certain offering, rather than how to implement them. Therefore, most patterns of this category are abstract and do not define measurable metrics that allow to check runtime behavior.

The *Cloud Environment* patterns describe how cloud environments can be offered according to the cloud service models Infrastructure as a Service (IaaS) and Platform as a Service (PaaS). *Elastic Infrastructure* and *Elastic Platform* are not measurable as they are too abstract and do not define detailed metrics. On the other hand, *Availability Patterns* describe a metric that allows checking during application runtime. Considering the view of an IT architect, it is valuable to check whether designed components actually adhere to the defined availability. Hence, I consider the patterns *Node-based Availability* and *Environment-based Availability* as suitable patterns for further analysis.

Patterns of the subcategory *Processing Offerings* describe how cloud functionality should be hosted and how large workload can be processed with high performance. Even though those patterns are important to understand, it is not meaningful to describe these patterns

by formal constraints, as there are no measurable metrics defined. Hence, a checking at runtime is not feasible, which is why this subcategory is omitted from further analysis.

Storage Offering patterns describe different variants of storing data in the cloud. The patterns *Block Storage*, *Blob Storage*, *Relational Database*, and *Key-Value Storage* describe offerings that are not subject to change during runtime. The patterns *Strict Consistency* and *Eventual Consistency* describe how storage offerings can handle data consistency according to the CAP theorem [28]. Strict consistency is measurable as the pattern defines a ratio between the number of replicas accessed by read and write operations. However, checking data consistency is usually ensured by the database management system or by the cloud provider [24]. In the case of integrating different cloud providers or legacy applications in static data centers, checking consistency becomes dependent on the integration environment. This requires a specific implementation tailored to the integration scenario, which is why the pattern *Strict Consistency* is not considered to be suitable for further analysis. *Eventual Consistency* has a relaxed consistency view and, thus, does not define a specific ratio that has to be satisfied. It is not possible to decide whether data will be consistent in the future or not, which is why this pattern is neither considered for further analysis.

Patterns of the subcategory communication offerings describe how data can be exchanged cloud internally and externally. The fundamental communication offerings are described by the patterns *Virtual Networking* and *Message-oriented Middleware*. These two patterns explain the basic architecture and mechanisms to e.g. exchange messages between components. Therefore, the patterns are very abstract and do not define measurable properties that facilitate runtime checking. The remaining patterns of this subcategory describe different delivery approaches, that can be applied to communication offerings. A detailed analysis, whether the intended delivery modes are implemented correctly, would require detailed investigation of different middleware technologies and exceed the scope of this work.

4.1.3 Cloud Application Architecture Patterns

The general structure of cloud applications and application components is described by patterns of the *Cloud Application Architecture* category. These patterns cover fundamental architectural styles that utilize cloud offering patterns as runtime environment to build cloud-native[1] applications. Further patterns of this category cover descriptions of cloud application components, cloud service sharing strategies, and cloud integration approaches to enable implementation and usage of hybrid clouds.

The fundamental patterns *Loose Coupling* and *Distributed Application* should be known by architects and developers when building cloud-native applications. Loose coupling is an essential concept in cloud computing that aims to reduce the assumptions two parties

[1] Cloud-native applications embrace the essential cloud properties as defined by the cloud computing definition of NIST [40]

make about each other. The according pattern forms the basis for distributed applications, however, it is not measurable as it describes abstract principles. The *Distributed Application* pattern presents three solution approaches for its implementation, i.e. layer-based decomposition, process-based decomposition, and pipes-and-filters based decomposition. Layer-based decomposition defines exact component access restrictions that must be met to implement this decomposition style. Components only are allowed to access components of the same layer or one layer below. This access restriction can be checked for consistency at design time by analyzing the relations between components. Process-based decomposition describes a specific order style in which components can be called. This order is for example defined by a business process. The process-based decomposition style itself does not define general access constraints, like the layer-based decomposition style does. The same applies to the pipes-and-filters-based decomposition, as it describes an abstract processing style that requires a process model to verify access restrictions. I do not consider the layer-based decomposition style for further analysis, as the metrics of this pattern (e.g. layer-number) do not vary at runtime, plus it offers the possibility to check the consistency at design time by static analysis. The two other decomposition styles itself do not define detailed restrictions, as they require a separate process model to verify the solution by constraints and are not considered in further analysis.

The subcategory *Cloud Application Components* contains patterns that describe how single components of a cloud application can be implemented. I.e. these patterns define the functional requirements of individual components and processors that can be used in cloud applications. I do not see these patterns as suitable for further analysis, as a serious consistency check would require detailed source code analysis and, thus, is too specific to implement. For these kinds of patterns I rather suggest checking the component functionality by Unit-Tests. The last pattern of this subcategory, i.e. the *Multi-Component Image* pattern, is not considered suitable as it describes an abstract process that does not define measurable metrics. *Multi-Tenancy* and *Cloud Integration* patterns do not define clear conditions that can be measured substantially, which is why patterns of these categories are omitted from further analysis.

4.1.4 Cloud Application Management Patterns

The category of *Cloud Application Management* contains patterns describing components and processes that automatically manage cloud-native applications. These patterns explain how to deal with different runtime metrics, such as number of IT resources, workload experienced by components, or size of data processed, to decide how the cloud application must react.

The *Provider Adapter* pattern describes a component that encapsulates provider interfaces and maps those to a unified interface to achieve separation of concerns. A provider adapter must conform to the architecture described in the pattern, however, the description does not show measurable runtime metrics, hence I do not consider this pattern as suitable for further analysis. The *Managed Configuration* pattern describes how scaled-out application components can manage their configurations. Two approaches are described to implement

this pattern, either the newly launched instances pull the configuration from a storage, or they receive new configurations automatically. These approaches can be expressed by a business process and, thus, this pattern can be checked by the approach of Mulo et al. [41]. Additionally, in the pattern description there are no measurable metrics defined which does not qualify this pattern according to the defined criteria. The patterns *Elasticity Manager*, *Elastic Load Balancer*, and *Elastic Queue* manage the scaling of cloud application components according to varying metrics, as e.g. experienced utilization, number of application accesses, and number of messages. These patterns contain measurable properties that vary at runtime, consequently I see all three patterns as suitable, as they conform the selection criteria. Furthermore, the *Watchdog* pattern is a suitable pattern, because it describes how to cope with application failures by monitoring runtime properties and defining clear conditions on how to react on insufficient availability.

The subcategory *Management Process* contains patterns describing how cloud applications can cope with runtime challenges. These patterns are based on patterns of the previous subcategory *Management Components* and describe abstract management processes expressed in Business Process Model and Notation (BPMN) [10] language. While it would be possible to define formal constraints for these patterns, they will be omitted from further analysis. I rather refer to the related work of Mulo et al. [41] to check SOA compliance when implementing and checking these patterns.

4.1.5 Composite Patterns

Patterns of this category cover how patterns of previous categories can be combined to describe cloud applications. There are two subcategories, i.e. *Native Cloud Applications* that cover fundamental structures of distributed applications, and *Hybrid Cloud Applications* that explain different structures of hybrid cloud applications refining the fundamental composite patterns.

The *Two-Tier Cloud Application* and *Three-Tier Cloud Application* patterns both are based on the *Distributed Application* pattern and describe the separation of cloud applications into two, and three logical tiers, respectively. Because these pattern describe a refinement of *Distributed Application* and, thus, do not conform the selection criteria, they are not selected as suitable candidates for further analysis. The pattern *Content Distribution Network* describes that content should be distributed among different physical locations. As this pattern explains a rather abstract structure and does not define measurable metrics, this pattern is not considered to be suitable, either.

Patterns in the subcategory *Hybrid Cloud Applications* are described in an abstract nature and do not define strict properties that are measurable at runtime. These patterns cannot be expressed reasonable by semantic constraints, which does not qualify them as suitable candidates according to the defined selection criteria.

Pattern Name	Pattern Category
Static Workload	Cloud Computing Fundamentals
Periodic Workload	Cloud Computing Fundamentals
Continuously Changing Workload	Cloud Computing Fundamentals
Node-based Availability	Cloud Offerings
Environment-based Availability	Cloud Offerings
Elasticity Manager	Cloud Application Management
Elastic Load Balancer	Cloud Application Management
Elastic Queue	Cloud Application Management
Watchdog	Cloud Application Management

Table 4.1: Selected Patterns and according pattern categories

4.1.6 Selected Cloud Computing Patterns

Table 4.1 summarizes all patterns that conform to the defined selection criteria. Consequently, a detailed analysis of these patterns can be conducted solely based on the pattern description, without the necessity to interpret too many facts into the description. Therefore, the patterns are analyzed in more detail in the following sections of this chapter. The aim is to extract detailed indicators that make it possible to recognize whether a pattern is actually implemented as defined in the application architecture design. The constraint language and framework, which are used in the presented method of Chapter 3, are explicitly designed and developed to dynamically check the consistency of all patterns that are selected in this section.

4.2 Metrics

In this section further analysis is conducted based on the selected patterns found in Section 4.1. The intention of this analysis is to extract measurable metrics that allow to precisely describe the structure and semantics of cloud computing patterns. For this analysis the pattern elements *Intent*, *Context*, *Solution*, and *Result* are investigated, as these pattern elements contain detailed explanations of the environmental structure and behavior. The Goal-Question-Metric (GQM) [5, 6] approach is suitable for this analysis as it allows to collect metrics from problems, goals, and questions [39]. Accordingly, GQM consists of three steps: (1) define the relevant goals, (2) extract questions for every goal that have to be clarified, (3) determine metrics for every question that contribute towards answering the question.

Definition 4.2.1 (Goal of the Metric Analysis)
Finding measurable metrics of selected cloud computing patterns, such that these patterns can be expressed by formal constraints.

The first step of the GQM analysis is addressed by expressing the goal in Definition 4.2.1. In the following subsections, steps two and three are applied to the patterns which are selected in Section 4.1.

4.2.1 Workload Patterns

Application workload patterns describe different workloads that are caused by users accessing the cloud application or the automatic processing of jobs by components. In the workload pattern descriptions, the term workload refers to the utilization of IT resources. To decide which workload is experienced, it is necessary to figure out how the workload can be measured accurately.

Definition 4.2.2 (GQM Question - User Workload)
How much workload is caused by the users directly?

Hence, the first question for workload patterns is defined according to the GQM approach in Definition 4.2.2. Users accessing the application produce workload, as the user requests are being handled by the application components. Accordingly, we want to measure the communication traffic by the number of requests that users send to an application. However, the number of requests alone does not allow the comparison of workload caused by users at different points in time. We need to detect changes over time, which is why the number of requests must be set in relation to time. The granularity of time is not defined in the pattern. There are applications that might be accessed by thousands of users per second, as well as applications that have only a few requests per day. Hence, the metric is defined as *requests per time unit*, where the time unit can be any multiple of a millisecond, also seconds, minutes, hours, days, etc..

Definition 4.2.3 (GQM Question - Workload on Virtual Machines)
How high is the workload on virtual machines?

In PaaS clouds, applications are deployed on platforms where the underlying infrastructure is usually not visible to the customer and no information about the underlying infrastructure can be extracted. Consequently, in this case it is better to focus on IaaS clouds, where runtime information about virtual machines is visible to the customers of the cloud, which is addressed by the question in Definition 4.2.3. In the workload patterns, the processing load on virtual machines is expressed by *Central Processing Unit (CPU) utilization*, *memory utilization*, and *disc space*.

Definition 4.2.4 (GQM Question - Workload on Storage Offerings)
How high is the amount of data stored in storage offerings?

A further GQM question is defined in Definition 4.2.4, focusing on storage offerings. For storage offerings that do not define a maximum storage limit, the workload can be expressed by the size of data that is stored on the storage offering. This amount is commonly

expressed in the unit byte, which leads to the metric *amount of occupied bytes*. To define a metric that is comparable among different types of storage offerings with a storage limit, it is of importance to set the occupied storage of an offering in relation to the maximum amount that is available, if defined. Hence, another metric to describe the workload of storage offerings is defined as *percentage of occupied bytes*, regarding total available bytes.

4.2.2 Availability Patterns

The patterns *Node-based Availability* and *Environment-based availability* describe different availability definitions for cloud environments. According to the ISO 25010 standard [33], availability is defined as the degree to which a system or component is operational and accessible when required for use. Correspondingly, the first pattern refers to availability that has to be assured for every individual component of the cloud application, while the second pattern defines availability that considers all components of an application combined.

Definition 4.2.5 (GQM Question - Availability on Individual Components)
What is the availability for individual cloud components?

The first question that must be clarified is expressed in Definition 4.2.5. As explained in the *Context* section of the pattern *Node-based Availability*, availability must be defined by certain conditions and a timeframe has to be expressed to which the conditions must hold. The conditions for *Node-based availability* are that an application must be reachable and perform its functions as advertised. This can be verified by heartbeat messages that are send by a component to a monitor. By defining a timeframe in which heartbeats must be received by the monitor, it can be verified whether an application is still reachable or not. The same applies to test messages that are continuously sent to a component, to check if the component responses with expected results. Thus, a metric can be defined as *amount of heartbeats per time unit* and *amount of correct test results per time unit*, respectively.

Further metrics regarding the definition of availability are expressed by the duration of unavailability, as described by Leymann [37] or Ludewig [39]. Mean Time Between Failures (MTBF) defines a metric that describes the average time that a component or virtual server is available between two consecutive failures [24]. Mean Time To Recovery (MTTR) defines a metric for unavailability that describes the time it takes to recover a failed component to become available again after a failure [24].

Definition 4.2.6 (GQM Question - System Availability)
What is the availability for the entire system?

The second question for availability is defined in Definition 4.2.6. *Environment-based Availability* does not rely on the availability of individual nodes, instead the availability has to be expressed considering the whole cloud application. To compute the availability of the environment, the application can be tested for reachability and correct functionality, similarly to the tests for node-based availability. Thus, the metrics for environment-based

availability are the same as for node-based availability. But instead of testing single nodes, the application reachability and functionality as a whole must be checked via provided interfaces.

4.2.3 Management Patterns

Management component patterns describe architectural components that analyze the cloud application properties to manage other cloud application components automatically. The selected patterns of this category do not share the same properties as patterns in the previous subsections. Hence, the patterns are analyzed separately by applying the steps two and three of the GQM approach individually to each selected pattern of this category.

Elasticity Manager

The *Elasticity Manager* pattern describes how cloud applications can adapt to varying workload by scaling application components.

Definition 4.2.7 (GQM Question - Application Utilization)
How high is the workload utilization of the monitored cloud application?

Thus, the first question that must be clarified is defined in Definition 4.2.7. Equally to the discussion for workload patterns, we mainly focus on IaaS clouds, as PaaS service model providers usually hide utilization information of the hosting machines from the customer. In the case of IaaS the pattern describes utilization of virtual machine instances by the metrics *CPU utilization*, *memory utilization*, *disc space*, and *network I/O*.

Definition 4.2.8 (GQM Question - Functionality of the Elasticity Manager)
What indicates that the Elasticity Manager is working as expected?

A second question, defined in Definition 4.2.8, must be clarified to argue whether the scale operation is properly enforced. If the utilization rises, more component instances are required to handle the workload, and vice versa, less component instances are required. Adhering to the pattern description, a correct functionality of the pattern can be indicated by the number of required components. Because only a running component can handle workload, the pattern description is refined and, thus, the metric is defined as *number of running component instances*.

Elastic Load Balancer

Equivalently to the *Elasticity Manager*, the *Elastic Load Balancer* defines the number of running instances according to the experienced workload of the application.

Definition 4.2.9 (GQM Question - Elastic Load Balancer Monitoring)
What are the properties monitored by the Elastic Load Balancer?

We first define a question to identify trigger metrics in Definition 4.2.9. To gather monitored information, this pattern checks the load balancer component to extract the number of synchronous accesses to the application. The number of requests has to be set into relation to a time value, which is why the metric will be defined as *number of synchronous requests per time unit*, where time unit can be any multiple of a millisecond, e.g. second, minute, hour, etc.. In addition to the information extracted from the load balancer, the utilization of components can be considered to calculate more detailed capacity planning of the *Elastic Load Balancer*. That means, *CPU utilization*, *memory utilization*, and *disc space* are additional metrics that can be considered for this pattern.

Definition 4.2.10 (GQM Question - Functionality of the Elastic Load Balancer)
What indicates that the Elastic Load Balancer is working as expected?

A further question that must be clarified, regarding correct functionality indicators of the pattern, is defined in Definition 4.2.10. The *Elastic Load Balancer* must scale application component instances according to the monitored metrics. Hence the indicator on a correct functionality of this pattern is the same as for the *Elasticity Manager*, i.e. the *number of running component instances*.

Elastic Queue

This pattern considers the asynchronous accesses via messaging to an application, to define the required number of component instances. It is necessary to consider two different kinds of metrics, i.e. one being monitored by the management component and, the other which is the result of the pattern management operations.

Definition 4.2.11 (GQM Question - Elastic Queue Monitoring)
What are the properties monitored by the Elastic Queue?

Consequently, the first question for this pattern is defined in Definition 4.2.11. The most important monitoring information is the *number of messages* that a queue contains. But the pattern also describes that information about the messages itself can be an indicator on how many scaled-out components are required. Information about the messages is expressed as message priority, which can be expressed on a nominal scale with values such as "informational", "critical", "status event", or on an ordinal scale with numbers within a specific range, e.g. 0 (indicating low priority) to 9 (indicating high priority), as implemented for example by the Java Message Service [30]. By using at least an ordinal scale, the metric *average priority* in the queue can be determined, which can then be used to define more fine-grained computations for the management reaction. To compute the number of required instances in even more detail, the *Elastic Queue* can extract component utilization information, such as *CPU utilization*, *memory utilization*, and *disc space*. The

pattern description also contains environmental information that can be considered before provisioning resources, which is defined as *resource price*.

Definition 4.2.12 (GQM Question - Functionality of the Elastic Queue)
What are the properties which indicate that the Elastic Queue is working as expected?

A further question that must be clarified, regarding correct functionality of the pattern, is defined in Definition 4.2.12. The *Elastic Queue* is responsible for scaling the application components according to the monitored metrics. I.e. an indicator on a correct functionality of this pattern is described as the *number of running component instances*.

Watchdog

The *Watchdog* pattern describes a component that automatically copes with failures of component instances by monitoring various information sources.

Definition 4.2.13 (GQM Question - Watchdog Monitoring)
What are the properties monitored by the Watchdog?

As this pattern monitors properties of component instances, the first question for this pattern is accordingly expressed in Definition 4.2.13. The pattern defines multiple information sources that can be monitored, e.g. periodic heartbeats and test requests are mentioned to check the state of a component. This indicates the availability of a component and can be expressed, similar to the definition for availability patterns, by the metric *heartbeats per time unit* and *number of correct test results per time unit*, respectively. Further information sources that are described as component status indicators are, the network connectivity (e.g. *network I/O*) or other utilization information like *memory utilization* of a component. According to the *Variants* section of the pattern, the *operation time* can also be monitored and used as a metric.

Definition 4.2.14 (GQM Question - Functionality of the Watchdog)
What are the properties which indicate that the Watchdog is working correctly?

Additionally, it must be clarified what indicates that the pattern is working correctly, which is expressed by the question in Definition 4.2.14. A correct working *Watchdog* implementation replaces failed components and provisions a new instance of that component. The pattern usually monitors component groups that have a certain availability. I.e. the metric *availability* of a component group can indicate if the *Watchdog* is working properly. A further metric that indicates the right operation of the pattern is the *number of running component instances*, as the *Watchdog* has to keep the number of instances in the group at a certain level to maintain availability.

4.3 Constraints

In this section, every selected pattern of Section 4.1 is analyzed for constraints. The analysis is processed by analyzing the pattern elements *Intent*, *Context*, *Solution*, *Result*, *Known Uses*, and *Variants* for phrases that express pattern behavior, conditions, invariants, or boundaries. It explains what has to be checked to verify that the pattern is in fact adhering to its definition. Consequently, for every pattern, statements are defined that concisely describe constraints in the form of informal prose text. The pattern descriptions do not always define one strict behavior or single conditions, but rather explain various ways to implement the pattern and, therefore, for some patterns more than one constraint statement is defined. Additionally, for every pattern an example validation implementation, based on the metrics found in Section 4.2, is explained to elaborate comprehension of the constraint statements.

4.3.1 Static Workload

The *Static Workload* pattern describes that the workload of IT resources is characterized by a more-or-less flat utilization over time within certain boundaries. In the pattern solution description, it is explained that static workload does not change at all or only very little over time. This means that the pattern can be either expressed strictly, without allowing even minimal variations, or the pattern can be expressed by adding a granularity value to the constraint.

Definition 4.3.1 (Static Workload - Strict Constraint)
The experienced workload of every monitored application component must not vary at all times.

A strict constraint for static workload is expressed in Definition 4.3.1. However, this constraint can barely be satisfied for real world cloud applications, because there are factors like e.g., network connectivity, or human user behavior, that can vary over time and, thus, has an impact on the experienced workload. Furthermore, if the application is hosted in a public cloud there might be variations in resource utilization, because IT resources are provided to a very large customer group. Accordingly, it is more feasible to add a variable to the pattern constraint that defines the level of granularity for the pattern. Another option that can be applied to filter real world variations is to use aggregate functions, as e.g. the average function. A time range can be defined within which the application workload is aggregated and compared to the previous aggregation value. This leads to the constraint in Definition 4.3.2, which can be applied to verify static workload in real world cloud applications.

Definition 4.3.2 (Static Workload - Applicable Constraint)
The aggregated workload of all monitored application components, within a certain time, must not vary more than a predefined variation value from a defined reference workload.

A concrete variant of this constraint, using the metrics defined in Section 4.2, can be implemented, e.g. by calculating the average value of CPU utilization of all virtual machines over a certain period of time. Then, a reference workload value has to be selected, e.g. by using the first measurement of the average workload. The constraint is satisfied if every average workload measurement does not vary more than, for example, 3% from the initial reference workload.

4.3.2 Periodic Workload

Periodic workload is experienced when the application's resource utilization is growing at recurring time intervals and dropping again afterwards. This behavior can be described by utilization peaks at recurring time intervals.

Definition 4.3.3 (Periodic Workload Constraint)
Application components must experience a peak in workload utilization at recurring time intervals.

The constraint for the pattern *Periodic Workload* is described by Definition 4.3.3. A simple solution can be implemented by calculating the overall average base workload, then, a peak can be described as workload that crosses the average workload by a certain value. However, in case the workload increases on every virtual machine and scaling is considered, new instances of components can be provisioned. This would result in an overall workload decrease, as the workload is distributed between more components. Thus, building the average value is not beneficial when scaling is considered. Instead, when calculating the summarized value of the workload of all components, it is possible to detect absolute workload growth independent of scaling effects. Other ways to express utilization peaks can be calculated by utilizing algorithms on data sets, as e.g. presented by Palshikar in [48].

4.3.3 Continuously Changing Workload

Continuously Changing Workload describes applications that experience long term change in workload. This change must happen consistently in one direction, i.e. either the workload is continuously growing or continuously decreasing by a change rate. Mathematical functions can be used to express the change rate in greater detail, e.g. the change can be linear, exponential, logarithmic, etc.

Definition 4.3.4 (Continuously Changing Workload - Strict Constraint)
The experienced workload of every monitored application component must continuously grow / decrease by a certain change rate over time.

The first constraint for the pattern *Continuously Changing Workload* is expressed by Definition 4.3.4. This, however, is a very strict definition of the pattern that might not be

feasibly applicable for real world cloud applications, because of similar reasons as defined for the pattern *Static Workload* in Section 4.3.1. Small variations in experienced workload might occur for single components, or even for the whole application. Considering this, it is more feasible to use aggregate functions, like for example computing the average component utilization of all components over a certain time period. However, in case the utilization of components increases, cloud applications might provision new resources to handle the growing workload. This can tamper workload measurements as the average value is dependent on the number of components in the system. Therefore, when using average aggregations and scaling is considered, the number of component instances must be considered in the calculation.

Definition 4.3.5 (Continuously Changing Workload - Applicable Constraint)
The aggregated workload of all monitored application components, within a certain time, must continuously grow / decrease.

The constraint in Definition 4.3.5 describes the pattern *Continuously Changing Workload* in a more feasible and generally applicable way as the definition before. A concrete variant of this constraint can be implemented, e.g. by summing up the used disc space of all storage offerings at certain periods of time. When considering continuously growing workload, the current measured summarized value must be greater than the previous measured summarized value.

4.3.4 Node-based Availability

Cloud providers usually define an availability value for the offered IT resources. For the pattern *Node-based Availability*, this value has to be verified for every individual node in the application, i.e. for every individual resource such as virtual machine instance, storage offering, middleware, etc. Customers of cloud offerings can verify the availability of their purchased resources through provided user interfaces or application programmable interfaces. Furthermore, customers may want to verify the availability of their own managed middleware or implemented applications. In this case, customers must implement their own availability checks, e.g. by requesting heartbeats, as explained in Section 4.2.2.

Definition 4.3.6 (Node-based Availability Constraint)
The availability of a specific node, or every individual node, has to be greater or equal than a predefined availability threshold.

Definition 4.3.6 defines a constraint for the pattern *Node-based Availability*. This constraint considers that every component is verified individually, however, it is also applicable to logic components that consist of multiple single components, as described in the pattern *Solution*. By considering availability for logically composed components or even the whole application, the individual availability values must be multiplied with each other. E.g. to verify node-based availability for virtual machines, the customer implements an application that sends ping requests every second to every virtual machine instance in the cloud, to

calculate the actual availability value. Then, this value can be compared to the predefined availability value to verify whether the constraint is compliant or not.

4.3.5 Environment-based Availability

Cloud providers offer an *Elastic Infrastructure* or an *Elastic Platform* that allows the deployment multiple application components. Individual components of the environment might not be available, however, the environment as a whole is still reachable and does execute the desired functionality. Therefore, it is not meaningful to consider the availability for individual nodes as done for the pattern *Node-based Availability*. Instead, the availability is expressed by the reachability and functionality of the environment, including all components.

Definition 4.3.7 (Environment-based Availability Constraint)
The availability of the cloud environment has to be greater or equal to a predefined availability threshold.

The cloud provider defines an availability value for the whole environment and, thus, the constraint for this pattern is expressed accordingly in Definition 4.3.7. This constraint can be implemented by, e.g. accessing the application every second via a provided interface to check reachability and functionality, which allows to calculate the availability as defined in Section 4.2.2. The availability then has to be compared to the predefined threshold value to verify if the pattern implementation is adhering the constraint.

4.3.6 Elasticity Manager

The *Elasticity Manager* pattern describes a management component that is responsible for scaling component instances according to the experienced workload. Scaling is necessary to ensure that resource utilization is maintained within defined thresholds. In the *Known Uses* section of the pattern it is explained that cloud vendors usually implement this pattern by provisioning or decommissioning component instances, whenever the application workload exceeds a predefined threshold. As the provisioning of new component instances requires time, it is more feasible to define a time interval within which a new instance has to be ready to process workload.

Definition 4.3.8 (Elasticity Manager - Procedural Constraint)
The number of application component instances has to be increased / decreased within a certain time interval, in case the application workload exceeds the defined upper / lower utilization threshold.

A constraint for the pattern *Elasticity Manager* is defined in Definition 4.3.8. This constraint focuses on the management functionality of the pattern, i.e. whether the scaling behavior is correct according to the pattern definition. However, it is also possible to have a

more general view on the pattern, by defining an equation based on the utilization and the number of components in the application. The utilization of all components can be aggregated to define a value representing all monitored components. This value can be compared to the current state of the scaled-out components.

Definition 4.3.9 (Elasticity Manager - General Constraint)
The number of running application component instances must be proportional to the aggregated utilization information of IT resources.

A constraint for the pattern that does not focus on the management process behavior, but rather verifies the system state by an equation is defined in Definition 4.3.9. Concrete variants of the constraints can be implemented by monitoring the metric CPU utilization of all virtual machines and calculating the average utilization value. Then, an upper threshold value of e.g. 80% and a lower threshold value of e.g. 20% can be defined. The constraint is satisfied whenever the average CPU utilization stays between the upper and lower threshold, or in case the threshold is crossed, and a new component instance is provisioned / decommissioned within a certain time interval. Another variant that focuses on the second constraint definition can be implemented, e.g. by monitoring network I/O of components, i.e. counting the network requests per minute for all components. Then, this value can be used to define an equation like: $rpm * c \leq \#component$, where rpm is the number of requests per minute, $c \in \mathbb{R}$ is a proportional constant, and $\#component$ is the number of running components. That means, the constraint is compliant when the number of network requests per minute multiplied with a constant, is less or equal to the number of running component instances.

4.3.7 Elastic Load Balancer

The *Elastic Load Balancer* pattern describes a management component that mainly monitors the load balancer of a cloud application. According to the number of synchronous accesses experienced by the load balancer, the amount of workload handling component instances is determined. In case the current number of component instances is not sufficient to handle the number of synchronous requests efficiently, the *Elastic Load Balancer* management component has to scale-out component instances to maintain efficient workload handling.

Definition 4.3.10 (Elastic Load Balancer - General Constraint)
The number of running application component instances must be proportional to the number of synchronous requests experienced by the load balancer component.

Accordingly, Definition 4.3.10 describes a general constraint for the pattern *Elastic Load Balancer*. Additionally, the *Elastic Load Balancer* management component can monitor utilization information of running component instances, to determine the number of required component instances more explicitly. That means, the *Elastic Load Balancer* first has to measure the number of synchronous accesses experienced by the load balancer, then, reflect the degree of capacity utilization of components and, finally, determine whether the

application can cope with the requests or needs to scale-out component instances. The application experiences under-provisioning of resources, if the utilization of IT resources is too high to be processed efficiently. In this case, the number of component instances must be increased, and in case the application experiences over-provisioning the number of component instances must be decreased. As already discussed for the *Elasticity Manager* pattern, real world factors like the time to provision new components must be considered. Thus, it is more feasible to additionally consider time intervals in the constraint definition.

Definition 4.3.11 (Elastic Load Balancer - Procedural Constraint)
The number of application component instances has to be increased / decreased within a certain time interval, in case the application utilization in relation to the number of synchronous accesses states that the application experiences under- / over- provisioning.

We define a constraint statement that reflects the procedural constraint in Definition 4.3.11. This constraint can be implemented by, e.g. extracting the number of requests per minute from the load balancer component, and calculating the average CPU utilization of all application components. Then, it can be determined how many concurrent requests one component instance can handle, to calculate the number of required instances. If the number of running instances is lower than the number of required instances, the management component has to provision further instances within a time interval of e.g. 1 minute, which satisfies the constraint.

4.3.8 Elastic Queue

The *Elastic Queue* is a pattern that describes a management component that adjusts the number of application component instances according to the number of asynchronous accesses via messaging. Hence, message queues have to be monitored by the management component to count the number of non-processed messages and define the number of required component instances to process the messages efficiently. If the number of available component instances is less than the number of required component instances, the application experiences under-provisioning of IT resources, or over-provisioning vice versa. Therefore, the management component has to scale-out components via operations, provided by the elastic infrastructure or elastic platform. To address the provisioning time, it is important that the constraint also defines a time interval within which the management component has to provision new instances.

Definition 4.3.12 (Elastic Queue - Procedural Constraint)
The number of application component instances has to be increased / decreased within a time interval, if the number of messages in a queue determines that the application experiences under- / over- provisioning.

The statement in Definition 4.3.12 expresses the described behavior as a constraint. The pattern *Solution* and *Result* elements, furthermore, describe additional properties that can be considered when determining the number of required instances. Similarly to the *Elastic*

Load Balancer, the *Elastic Queue* also can consider the current utilization of the application, to decide whether the current number of components is capable of efficiently handling the number of messages. Another factor described in the pattern is environmental information, as provisioning new resources raises the runtime costs of an application. If the price of provisioning more cloud resources is not economically profitable compared to the benefit, it might be better to maintain the current level of workload handling instances or even scale-in the application. Additionally, the management component can postpone less critical messages if the provisioning of new resources is not profitable in the current situation, meaning the *Elastic Queue* should also consider the message types.

Definition 4.3.13 (Elastic Queue - General Constraint)
The number of component instances has to be proportional to the number of messages in a queue, the type of messages, the application utilization status, and the environmental cost factors.

Considering all factors of the pattern along with the number of messages in a queue, a more general defined constraint is expressed in Definition 4.3.13. The pattern can be implemented by monitoring the queue of a messaging platform and determining the number of required instances according to the current CPU utilization of application components. Then, an equation can be established to set the number of required instances in relation to the number of messages and CPU utilization. If the equation is violated, the *Elastic Queue* component has to provision / decommission an instance within e.g. 1 minute.

4.3.9 Watchdog

The *Watchdog* pattern describes a management component that automatically copes with failing application component instances, if the availability is insufficient according to a predefined value. The *Solution* element describes the main task of the pattern as, monitoring components and replacing them in case of failures. Furthermore, the time to provision a new component can be additionally considered, to make the constraints more feasible.

Definition 4.3.14 (Watchdog - Procedural Constraint)
Every failing application component instance has to be replaced by a new instance within a certain time.

A definition that expresses constraints for the pattern *Watchdog* is defined in Definition 4.3.14. This constraint describes the pattern from a procedural point of view, however it is also possible to define a more general constraint, e.g. as mentioned in the *Result* section of the pattern, by defining a minimum amount of running component instances to ensure a certain degree of availability for the application.

Definition 4.3.15 (Watchdog - General Constraint)
The number of running application component instances has to be greater or equal to a certain predefined threshold.

A further constraint for the pattern is defined by Definition 4.3.15. Additionally, a variant of the pattern is mentioned in the *Variants* section, where the *Watchdog* is applied to monitor the runtime of application components and replace instances if they exceed a certain runtime. This is especially beneficial, if the components are not suitable for long operation times or if they have never been tested over a long period of time.

Definition 4.3.16 (Watchdog - Variant Constraint)
Every monitored component instance has to be replaced after its operation time exceeds a predefined threshold.

Thus, we define this pattern variant in the constraint statement of Definition 4.3.16. The pattern can be implemented by sending heartbeats to application components in order to detect whether the instances are available or not. Whenever the *watchdog* component detects that an application failed, it has to provision a new instance to fulfill the constraint. The second constraint can be fulfilled by counting the number of running component instances and checking, whether the actual number is greater or equal to a predefined value. The last constraint can be verified by measuring the constant uptime of every component instance and checking whether the uptime is less than a predefined threshold. If components exceed the maximum operation time they have to be replaced by a new instance.

5 Constraint Language

This chapter describes the development of the constraint language, which is used for the formal documentation of cloud computing pattern constraints. The pattern analysis of Chapter 4 is used to define requirements on the language. To formally specify the language, a meta-model is defined and complemented with the definition of syntax and semantics. This enables to execute Step 2 of the defined method in Chapter 3, as the constraints of Chapter 4 can be expressed formally and without ambiguities. Eventually, the tooling for the language is explained, which also includes the mapping definition between language elements and EPL statements, corresponding to Step 3 of the defined method.

5.1 Requirements

The findings of the analysis in Chapter 4 are used in this section to define requirements for a formal constraint language that expresses the structure and semantics of cloud computing patterns. Metrics of cloud computing patterns have to be considered in mathematical expressions. Hence, those expressions consist of literals of different data types and are connected by mathematical operators. Furthermore, it is necessary that the language supports the expression of aggregate functions, as also utilized in Section 4.2 and in Section 4.3, for aggregating properties of application components. As cloud applications are composed of various different components, it is necessary to restrict defined expressions to a certain context. Within a defined context, an expression either indicates whether the cloud application is compliant to the design or violating patterns. This indication has to be provided live in reaction to runtime changes of the application and, therefore, it is required that the language is able to handle real-time status updates, e.g. in the form of events. In Section 4.3 there are patterns that can be expressed as a process, which is why the language must support the definition of event sequences. Additionally, for some patterns, the definition of time boundaries is crucial for the fulfillment of constraints, thus, including time intervals is a further requirement on the language. Another requirement on the languages is being able to clearly arrange the definition of pattern constraints. That means, for constraint comprehension, it is important to define interim statements of mathematical expressions and event sequences, that can be used in a compliance or violation constraint of the pattern. As most of the pattern constraints in Section 4.3 have variables, like e.g. thresholds or variations, it is necessary for a suitable language to support the definition of placeholder variables.

All requirements (RQs) on the cloud computing pattern constraint language are listed in the following enumeration:

RQ 1: Assignment of constraints to a certain pattern
RQ 2: Declaration of structural components
RQ 3: Definition of mathematical expressions
RQ 4: Usage of aggregate functions
RQ 5: Restriction of constraints to a context
RQ 6: Indication of pattern compliance or pattern violation
RQ 7: Definition of event sequences
RQ 8: Restriction of events to a time interval
RQ 9: Declaration of interim structures and semantics
RQ 10: Definition of placeholder variables

5.2 Meta-Model

The meta-model defined in this section addresses the required elements the constraint language must contain to satisfy the defined requirements of Section 5.1. Figure 5.1 shows the high-level meta-model view of the constraint language expressed as UML class diagram. Elements representing details are omitted in this view and explained separately to contain clarity for the main elements.

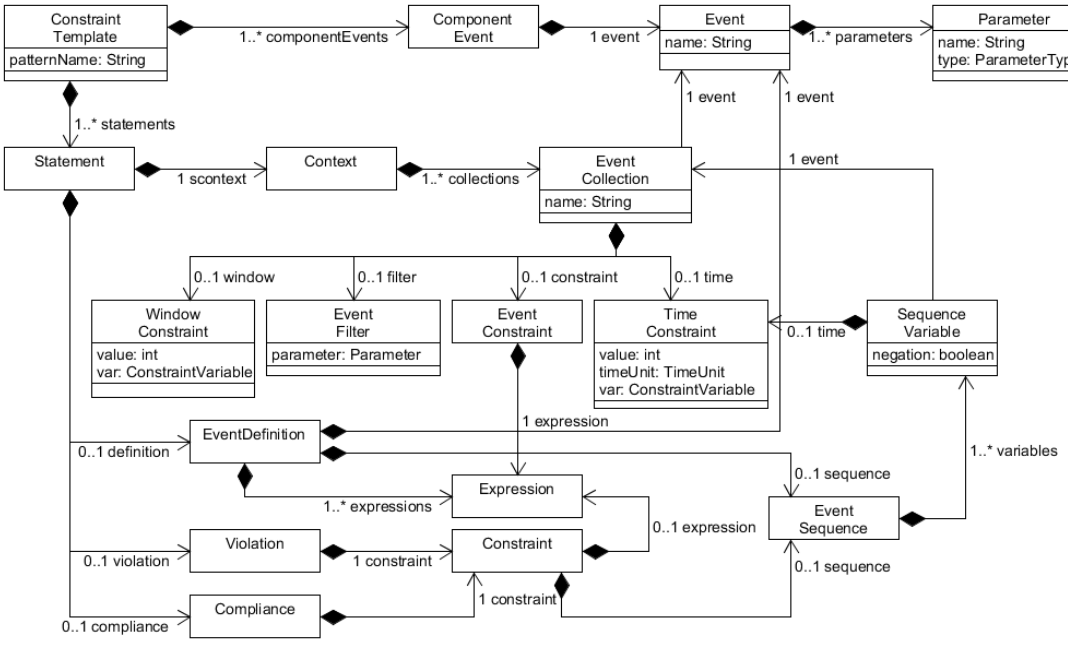

Figure 5.1: High-Level Meta-Model View of the Constraint Language

The first requirement RQ 1 is addressed by the *ConstraintTemplate* root element that holds the name of the pattern to be addressed and contains structural and semantic constraints. Structural constraints are defined by the *ComponentEvent* element (referring RQ 2), which allows the description of initial events that can be used by all statements in the constraint

template. A *ComponentEvent* describes what kind of application components are considered in the constraint template. Hence, the *ComponentEvent* uses an *Event* element, which corresponds to the *ComponentType* element of the domain-model, in this correlation. Instances of the *Event* element are, thus, the dynamic representation of cloud computing components, which correspond to the *Component* element of the domain-model. *Event* elements have multiple *Parameter* elements to describe properties of the cloud component. Hereby, *Parameter* elements correspond to the *Attribute* elements of the domain-model. Every *Parameter* has a certain name and an according *ParameterType*, which can be an integer number, a floating point number, a string of characters, or a boolean value.

Besides *ComponentEvent* elements, a *ConstraintTemplate* can have multiple *Statement* elements, that are either structural constraints expressed as *EventDefinition*, or semantic constraints expressed as *Compliance*, *Violation*, or also *EventDefinition*. Regarding RQ 6, a *Violation* element indicates that whenever the corresponding *Constraint* evaluates to true, the pattern addressed in the *ConstraintTemplate* element is violated. Conversely, if the *Constraint* of a *Compliance* evaluates to true, it indicates that the pattern is adhering to the pattern definition. *Constraints* can be either defined by a mathematical *Expression* or by a sequence of events. Furthermore, the language allows the definition of composed events, that enable the expression of logical structures and the derivation of semantics in interim events, addressing RQ 9. This can be realized by the declaration of *EventDefinition* elements. An event can be either composed by the combination of events via a mathematical expression or by the definition of an event sequence (RQ 9).

An *EventSequence* defines an order of events that must occur, or explicitly not occur, to evaluate the element to true (RQ 7). The occurrence, or non-occurrence, of an event in the sequence, can be bound to a time restriction by the usage of a *TimeConstraint* element (RQ 8).

Every *Statement* contains a *Context* element, which defines the scope of events that apply to the *Statement* (addressing RQ 5). A context can be defined by multiple *EventCollection* elements, which are a collection of events of a certain type that is defined by *ComponentEvent* elements or by *EventDefinition* elements. Additionally, the language allows defining an *EventConstraint*, which restricts the collection of events to a certain expression that must evaluate to true. That means, an *EventConstraint* can only select events that have parameters with a certain value. By the usage of *EventFilters* it is possible to only consider the last event that is uniquely identified by a certain event parameter. The language also allows the usage of a *TimeConstraint*, to only collect events that occur within the defined time frame. Furthermore, a *WindowConstraint* allows to restrict the number of collected events to a certain value.

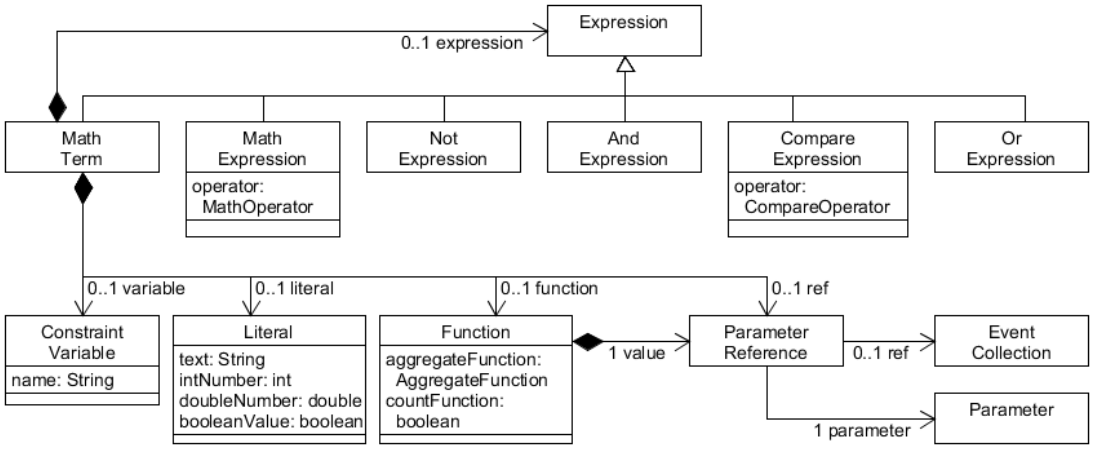

Figure 5.2: Meta-Model for Expressions Regarding the Meta-Model in Figure 5.1

5.3 Expressions

Expression elements in the high-level meta-model view (see Section 5.2) are further explained in this section to address RQ 3. Figure 5.2 shows the meta-model for expressions in UML class diagram notation.

An *Expression* element represents a mathematical expression that allows expressing basic mathematical algebra by combining numbers, literals, functions, and identifiers with mathematical operators. A *MathExpression* element allows the computation of a value by combining multiple *MathTerm* elements with *MathOperator* elements (cf. *MathOperator* terminals in Listing A.1). Hereby, the operations of the *MathOperator* are according to the standard rules of mathematical algebraic expressions and are grouped according to the grammar rules in Listing A.1.

Furthermore, it is possible to define Boolean algebraic expressions by using comparison operators, which can be grouped using *OrExpression* (logical disjunction, "OR" grouping), *AndExpression* (logical conjunction, "AND" grouping), and *NotExpression* (logical negation, "NOT" grouping) elements. A *CompareExpression* allows comparing numbers, literals, functions, and identifiers by using the compare operators as defined by the *CompareOperator* element in Listing A.1. These elements map the equation to a truth value which can be *true* or *false*. The grouping of those elements within an expression is indicated by the grammar definition in Listing A.1. All semantics for *CompareExpressions* correspond to the rules and laws of Boolean algebra.

MathTerm elements can contain exactly one element out of *ConstraintVariable*, *Literal*, *Function*, *ParameterReference*, or *Expression*. A *ConstraintVariable* element is a placeholder variable for a certain numerical or textual value, which corresponds to RQ 10. It is used to indicate thresholds and has to be replaced with an actual value when the constraint template is instantiated to check the constraints at runtime. *Literal* elements indicate either

an integer number "intNumber" $\in \mathbb{N}$, a floating point number "doubleNumber" $\in \mathbb{R}$, or a concatenation of characters "text" (see Listing A.1 for rules). *ParameterReference* elements allow referencing *Parameter* elements of *EventCollection* elements in an expression. For example, to reference a parameter "myParameter" of an event collection with name "myCollection", the *ParameterReference* element would be written as "myCollection.myParameter". The *Function* element allows the utilization of aggregate functions that can be applied to *EventCollection* elements, which refers RQ 4. Let E be the set of events for an event collection and P the set of event parameters. Then, p_e is a certain parameter $p \in P$ of an event $e \in E$, and $v(p_e)$ is the value of that parameter for the event e. Furthermore, $n \in \mathbb{N}$ is the amount of events in a collection, i.e. $n = |E|$, and e_i the i-th element in E, with $i \in [1..n]$. Possible functions for *Function* elements are defined as follows:

- "avg": Average function that can be applied to parameter values of data type "int" or "double", returning a "double": $avg(E, p) = \frac{1}{n}(\sum_{i=1}^{n} v(p_{e_i}))$.
- "sum": Function that sums up all values in a set. Can be applied to parameter values of data type "int" or "double", returning a "double": $sum(E, p) = \sum_{i=1}^{n} v(p_{e_i})$.
- "count": Function that returns the number of elements in an event collection: $count(E) = n$
- "product": Product function that multiplies all parameter values of data type "int" or "double", returning a "double": $product(E, p) = \prod_{i=1}^{n} v(p_{e_i})$.

5.4 Syntax and Dynamic Semantics

The definition of a grammar is a further step to formalize the constraint language as it clearly specifies how the language has to be utilized. Listing A.1 of Appendix A shows the grammar of the constraint language represented in Extended Backus-Naur Form (EBNF) notation. However, the grammar definition is long and verbose, which is why the critical meta-model elements are presented by syntax diagrams to improve comprehension. In the following, elements of the high-level meta-model will be described by syntax diagrams and, additionally, dynamic semantics are explained. Elements of the expression meta-model are not presented by syntax diagrams to maintain brevity and, thus, for those elements, it is referred to the grammar definition in Listing A.1.

5.4.1 ConstraintTemplate

The root element *ConstraintTemplate* indicates which pattern is addressed by the constraints. That means, all semantic and structural constraints, that are defined as a child element of the *ConstraintTemplate*, apply to the pattern that is mentioned in the root element. As depicted in Figure 5.3, the pattern name is addressed using an *ID* and enclosed by the terminal symbol ":". Following, at least one *ComponentEvent* must be defined, to indicate

which cloud application components are considered for the constraint. The end of a *ConstraintTemplate* is defined by at least one *Statement*.

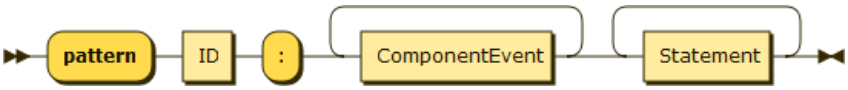

Figure 5.3: Syntax Diagram of the *ConstraintTempalte* element

5.4.2 ComponentEvent

A *ComponentEvent* describes a structural constraint on the application, indicating which cloud computing components are considered in a constraint template. *ComponentEvent* elements are indicated by the terminal "use", followed by the definition of one *Event* element, as shown in Figure 5.4. Finally, the element is finished with the end indicator terminal symbol ";".

Figure 5.4: Syntax Diagram of the *ComponentEvent* element

5.4.3 Event

An *Event* describes an entity that has a certain name and is defined by multiple parameters. The element has to be part of either a *ComponentEvent* or an *EventDefinition*, as it does not contain semantics on its own. As part of a *ComponentEvent*, it describes a component type in the cloud application. Otherwise, as part of an *EventDefinition*, it expresses either a logical structure that is composed of other structural constraints, or it expresses a semantic constraint which indicates application behavior. As depicted in Figure 5.5, The *Event* element definition starts with a unique event name using an *ID*, followed by a parameter list that is enclosed by the terminals "(" and ")". The terminal "," has to be included between *Parameter* elements if there are more than one.

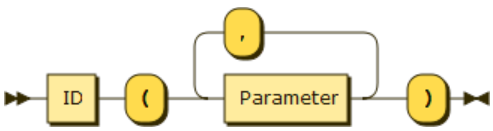

Figure 5.5: Syntax Diagram of the *Event* element

5.4.4 Parameter

Parameters contain key-value pairs that are used to describe the structure of events. Figure 5.6 shows the syntax for a *Parameter* element, which consists of a name, expressed as *ID*, and a *ParameterType* element. The *ParameterType* can be one of the terminals: "double" type for floating point numbers, "int" type for integer numbers, "string" type for concatenations of characters, "boolean" type for representing truth values *false* or *true*.

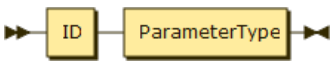

Figure 5.6: Syntax Diagram of the *Parameter* element

5.4.5 Statement

A *Statement* is the main element in a constraint template to describe semantic constraints that indicate pattern compliance or violation. Furthermore, a statement can be used to describe structural constraints by aggregating existing low-level or high-level events into complex events. For every statement, it is necessary to describe a context, which the defined structural or semantic constraint is restricted to. Figure 5.7 shows the *Statement* element syntax, which consists of exactly one *Context* element followed by either one *EventDefinition*, *Compliance*, or *Violation* element.

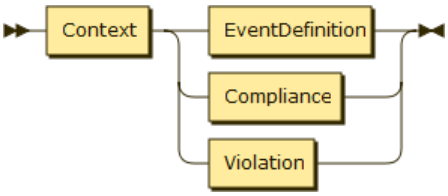

Figure 5.7: Syntax Diagram of the *Statement* element

5.4.6 Context

The *Context* describes which elements are considered for a certain statement. I.e. a statement can only consider event types that are explicitly collected in the context, hence, at least one event collection has to be declared. The syntax of *Context* elements is shown in Figure 5.8 and starts with the terminal "context", followed by a list of at least one *EventCollection* element. The list of *EventCollection* elements is separated by a "," terminal and closed by a ":" terminal.

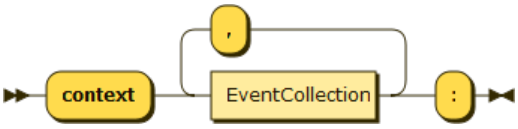

Figure 5.8: Syntax Diagram of the *Context* element

5.4.7 EventCollection

An event collection defines a set of events of a certain type, for which filters and constraints apply. Every time a new event is added to the collection, the corresponding *Constraint* element of the statement is triggered. It is possible to restrict the collection of events by using *EventConstraint* elements. Furthermore, it is possible to restrict the amount of collected events to a certain number or time interval, by using *WindowConstraint* or *TimeConstraint*, respectively. In case a *TimeConstraint* is given, the corresponding *Constraint* element of the statement is triggered additionally, whenever an event is removed from the collection as a consequence of the given time constraint. Eventually, it is possible to only consider the latest events which are uniquely identified by a certain parameter value, with the usage of an *EventFilter* element.

Figure 5.9 shows the syntax of *EventCollection* elements, which references an *Event* that is restricted by certain constraint and filter elements, and is assigned to a unique variable name. The first *ID* describes a variable name for the *EventCollection*, followed by the terminal "=" and another *ID* which is the name of an existing *Event* element. Following, there can be maximal one of each of the following constraint and filter elements given in the following order: *EventConstraint*, *EventFilter*, *TimeConstraint*, *WindowConstraint*.

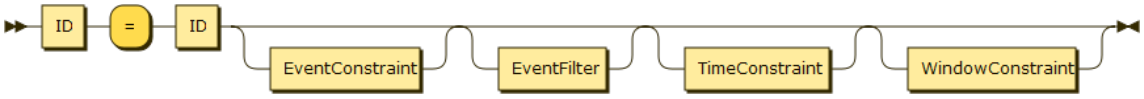

Figure 5.9: Syntax Diagram of the *EventCollection* element

5.4.8 EventConstraint

Constraints can be applied to event collections, to only consider events that match a certain expression regarding event parameter values. For example, by defining an event constraint that has the expression $cpu > 50$, only events are considered that have a value for the parameter "cpu", which is greater than the numerical value 50. As depicted in Figure 5.10, an *EventConstraint* consists of exactly one *Expression*, which is surrounded by a starting terminal "(" and an end terminal ")".

Figure 5.10: Syntax Diagram of the *EventConstraint* element

5.4.9 EventFilter

By defining filters, it is possible to only collect the latest event in a collection, which is uniquely identified by a certain parameter value. For example, by using a filter that is applied to an event parameter with name "vmID", only the latest events that contain this parameter with the same value are collected. That means, if an event collection contains an event with $vmID = 123$ and a new event with same parameter value arrives, the old event will be dropped and the new event will be inserted into the collection. Figure 5.11 shows the syntax diagram of *EventFilter* elements, which start with the filter indicator terminal "!" and are followed by a reference to a *Parameter* element name, expressed as *ID*.

Figure 5.11: Syntax Diagram of the *EventFilter* element

5.4.10 TimeConstraint

Time constraints can be defined to indicate that only events shall be contained in an event collection, that are not older than the provided time constraint value. For example, if a time constraint of 20 seconds is given, a collection dismisses events that are older than 20 seconds. Instead of giving a concrete time value, it is also possible to use a variable which can be set during consistency checking at runtime. The syntax of a *TimeConstraint* element is shown in the diagram in Figure 5.12. The element is indicated by the terminal "time" and either followed by an integer numerical value, expressed as *INT* non-terminal, or by a *ConstraintVariable* non-terminal. After the numerical value, a *TimeUnit* element indicates which time unit the constraint addresses. *TimeUnit* can be one of the following terminals: "milliseconds" for milliseconds, "sec" for seconds, "min" for minutes.

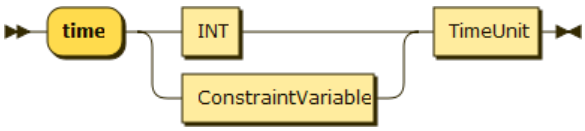

Figure 5.12: Syntax Diagram of the *TimeConstraint* element

5.4.11 WindowConstraint

The definition of window constraints allows restricting the amount of events in an event collection to a certain number. For example, if the window constraint value is set to 4, only the latest four events are kept in the collection. Instead of giving a concrete numerical value, it is also possible to use a variable which can be set during consistency checking at runtime. Figure 5.13 shows the syntax of *WindowConstraint* elements, which

are indicated by the start terminal "last(", either followed by an integer numerical value, or by a *ConstraintVariable* element and closed by the ")" terminal.

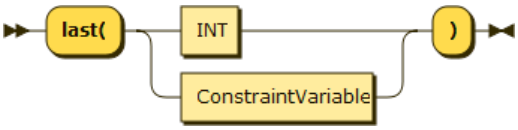

Figure 5.13: Syntax Diagram of the *WindowConstraint* element

5.4.12 EventDefinition

It is possible to aggregate component (low-level) events or composed (high-level) events into new events by using event definitions. The definition can use expressions to define the event parameter values. Additionally, the event definition can be bound to a certain sequence of events, i.e. whenever the defined sequence of events occurs, the definition creates a new event out of the expression values. The syntax for *EventDefinition* elements is described in Figure 5.14. The element starts with a definition indicator terminal "def:" and is followed by an *Event* element, which describes what kind of new *Event* element should be defined. The ":" terminal indicates the start of the list of *Expression* elements, which are separated by a "," terminal and closed by a ";" terminal. After the expression list, it is optionally possible to indicate that the *EventDefinition* depends on an event sequence. The start of an event sequence is indicated by the terminals "of" followed by "sequence:" and the definition of an *EventSequence* element.

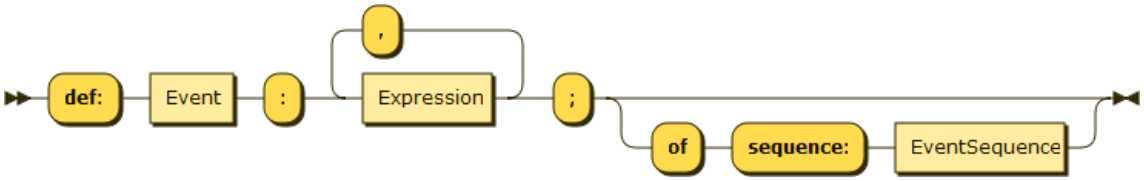

Figure 5.14: Syntax Diagram of the *EventDefinition* element

5.4.13 Compliance

The *Compliance* element enables the definition of semantic constraints, which indicate that the pattern addressed in the constraint template is compliant whenever the *Constraint* element evaluates to *true* and, thus, in this case, the application is consistent regarding the pattern. Figure 5.15 shows the syntax for the *Compliance* element. A terminal "compliance" indicates the start of the element, followed by one *Constraint* element.

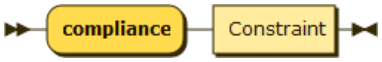

Figure 5.15: Syntax Diagram of the *Compliance* element

5.4.14 Violation

The *Violation* element defines a semantic constraint, which indicates that the pattern addressed in the constraint template is violated whenever the *Constraint* element evaluates to *true* and, thus, the application is inconsistent regarding the pattern. Figure 5.16 shows the syntax diagram of the *Violation* element, which starts with the terminal "violation", followed by one *Constraint* element.

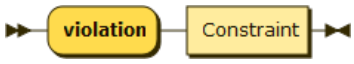

Figure 5.16: Syntax Diagram of the *Violation* element

5.4.15 Constraint

A constraint can either be expressed using an expression or by defining a sequence of events. The constraint evaluates to the truth value *true*, whenever the defined expression or event sequence evaluates to true. Hence, expressions for constraints must return a Boolean value. The *Constraint* element syntax is shown in Figure 5.17. A sequence constraint is defined by the two terminals "when" "sequence:", followed by an *EventSequence* element. A constraint with an expression is defined by the terminal "when:", followed by an *Expression* that is ended with the terminal ";".

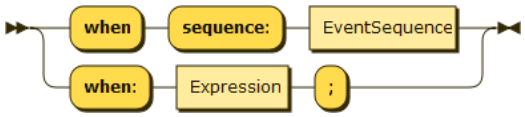

Figure 5.17: Syntax Diagram of the *Constraint* element

5.4.16 EventSequence

It is possible to define sequences of events by using sequence variables and defining a certain order for them. An event sequence evaluates to true when all events occurred in the given order, and as defined by the sequence variables. Figure 5.18 shows the syntax diagram of an *EventSequence*, which is a concatenation of *SequenceVariable* elements with a minimum amount of one *SequenceVariable*. The variables are separated by the terminal "->" and the sequence is ended by the terminal ";".

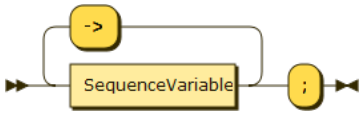

Figure 5.18: Syntax Diagram of the *EventSequence* element

5.4.17 SequenceVariable

Sequence variables are references to new events that are inserted into event collections. That means, whenever an event matches the event collection criterion, the sequence variable evaluates to the truth value *true*. It is possible to use negation logic for variables, which indicates that the event must explicitly not occur. Furthermore, time constraints can be used to bound the occurrence, or non-occurrence, respectively, to a certain time frame. E.g., when a time constraint of 10 seconds is given, the event referenced by the sequence variable must occur, or not occur, respectively, within 10 seconds. The syntax for a *SequenceVariable* element is shown in Figure 5.19. The element references the variable name of an *EventCollection* element, expressing it as an *ID*. It is also possible to describe the explicit non-occurrence of a certain variable in a sequence by using the leading terminal "not". Optionally, the occurrence of an event can be bound to a certain time value by using a *TimeConstraint* element.

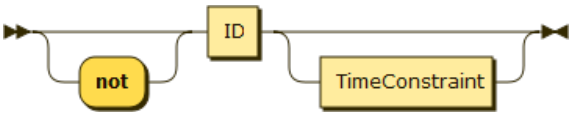

Figure 5.19: Syntax Diagram of the *SequenceVariable* element

5.5 Static Semantics

To provide a clear and unambiguous understanding and usage of a developed language, it is crucial to define clear semantics [59]. Well-formedness rules can be expressed by OCL constraints, that are applied to the meta-model of the language. In the following subsections, OCL constraints, according to the OCL specification [45], are defined regarding meta-model elements to formally specify how the language can be used.

5.5.1 Pattern Constraints

The *ConstraintTemplate* references which pattern is addressed by the structural and semantic constraints. Therefore, it is important that the pattern name must always be given, which is shown in Listing 5.1. Additionally, the constraint template must define at least one semantic constraint, that describes a violation or compliance of the pattern. That means, among

Listing 5.1 Static Semantics for Constraint Templates

```
context ConstraintTemplate
inv patternNameExists: not self.patternName->oclIsUndefined()
inv patternNameNotNull: self.patternName <> null
inv existsComplianceOrViolation: self.statements->exists(s | not
    s.violation->oclIsUndefined() or not s.compliance->oclIsUndefined())
```

Listing 5.2 Static Semantics for Statements

```
context Statement
inv varsContextDefinition:
    self.definition.sequence.variables->forAll(var |
    self.scontext.collections->exists(event | event.name =
    var.event.name))
inv varsAmountContextDefinition:
    self.definition.sequence.variables->size() =
    self.scontext.collections->size()
inv varsContextViolation:
    self.violation.constraint.sequence.variables->forAll(var |
    self.scontext.collections->exists(event | event.name =
    var.event.name))
inv varsAmountContextViolation:
    self.violation.constraint.sequence.variables->size() =
    self.scontext.collections->size()
inv varsContextCompliance:
    self.compliance.constraint.sequence.variables->forAll(var |
    self.scontext.collections->exists(event | event.name =
    var.event.name))
inv varsAmountContextCompliance:
    self.compliance.constraint.sequence.variables->size() =
    self.scontext.collections->size()
inv onlyOneExists: not self.definition->oclIsUndefined() xor not
    self.violation->oclIsUndefined() xor not
    self.compliance->oclIsUndefined()
```

all statements, there must be at least one statement that has a *Violation* or *Compliance* element.

5.5.2 Statements

Statement elements allow the definition of structural and semantic constraints by using sequences of events. According constraints are defined in Listing 5.2. The *SequenceVariable* elements that can be used in an *EventSequence*, refer to the *EventCollection* elements that are defined in the context. Consequently, for every variable used in an event sequence,

there must exist an according event in the context. Additionally, all event collections must be used in a sequence and it is not allowed to define unused event collections. This applies to all sequences used in either *EventDefinition*, *Violation*, or *Compliance* elements. Other than for OCL, it is not allowed to define more than one sub-element besides the *Context* for a *Statement*. This is due to the strict assignment of context collections and event variables in the statements, to not mix up associations. Hence, for every statement there must be either one of *EventDefinition*, *Violation*, or *Compliance*.

5.5.3 Unique Names

The context definition is crucial to describe what structural constraints are considered for a statement. Thus, it is necessary that all variables must be unique and that the names must actually be given. The same applies to the parameters of events, to not allow ambiguities when addressing event parameters. That means, the name of *Parameter* elements must be unique for every *Event*. Static semantics for unique names are shown in Listing 5.3.

Listing 5.3 Static Semantics for Names

```
context Context
inv uniqueNames: self.collections->isUnique(collection |
    collection.name)

context EventCollection
inv varName: not self.name->oclIsUndefined() and self.name <> null

context Event
inv: self.parameters->isUnique(parameter | parameter.name)

context ConstraintTemplate
inv uniqueEvents: self.componentEvents->collect(ce |
    ce.event)->union(self.statements->collect(s |
    s.definition.event))->isUnique(event | event.name)
```

Among all structural constraints there is no ambiguity allowed, meaning, there must be uniqueness in the definition of event names. *Event* elements are created when new *ComponentEvent* elements or new *EventDefinition* statements are given. Hence, the union of all *Event* elements of *ComponentEvent* elements and *EventDefinition* elements must be checked for unique names.

5.5.4 Event Definitions

When defining new events, it is possible to define multiple parameters. The static semantics for event definitions are shown in Listing 5.4. Each of the parameters must be defined by a certain expression, which is why it is necessary that the amount of *Expression* elements

Listing 5.4 Static Semantics for Event Definitions

```
context EventDefinition
inv amounts: self.event.parameters->size() = self.expressions->size()
```

used in an *EventDefinition* is equal to the amount of *Parameter* elements used by the *Event* element.

5.5.5 Collection Filters and Constraints

Listing 5.5 Static Semantics for Collection Filters and Constraints

```
context TimeConstraint
inv oneValue: not self.value->oclIsUndefined() xor not
    self.var->oclIsUndefined()
inv timeUnit: not self.timeUnit->oclIsUndefined()

context WindowConstraint
inv valueExists: not self.value->oclIsUndefined() xor not
    self.var->oclIsUndefined()

context EventFilter
inv paramExists: not self.parameter->oclIsUndefined()
```

It is possible to restrict *EventCollection* elements with filters and constraints. Consequently, it is necessary to define values for the filters and constraints, which can be concrete numerical values, or references to other elements like e.g. *ConstraintVariable*. Listing 5.5 shows the constraints for filters and constraints on event collections.

5.5.6 Expressions

Static semantics for expressions are defined regarding the expression meta-model in Figure 5.2. The OCL constraints are shown in Listing 5.6, which express that for one *MathTerm* element only one of the available sub-elements can be used. Furthermore, a *Literal* element can only express one of its attributes, i.e. either text, integer, floating point number, or Boolean value. For *ConstraintVariable* elements and *Function* elements, it is required that the according attributes must be given. When using the *count* function in a statement expression, the context must have exactly one *EventCollection* element, such that it is clear on which collection the count function is applied to. For the constraint statement regarding the *count* function, the "allOwnedElements" operation is utilized according to the definition in the UML specification [46].

Listing 5.6 Static Semantics for Expressions

```
context MathTerm
inv onlyOneExists: not self.variable->oclIsUndefined() xor not
    self.literal->oclIsUndefined()
xor not self.function->oclIsUndefined() xor not
    self.ref->oclIsUndefined() xor not self.child->oclIsUndefined()

context Literal
inv onlyOneType: not self.text->oclIsUndefined() xor not
    self.intNumber->oclIsUndefined() xor not
    self.doubleNumber->oclIsUndefined() xor not
    self.booleanValue->oclIsUndefined()

context ConstraintVariable
inv name: not self.name->oclIsUndefined() and self.name <> null

context Function
inv fun: not self.aggregateFunction->oclIsUndefined() xor
    self.countFunction

context Statement
inv countContext:
    self.allOwnedElements()->selectByType(Function)->exists(f |
    f.countFunction ) implies self.scontext.collections->size() = 1
```

5.6 EPL Statement Generation

When creating Domain Specific Languages, it is important to provide tooling to make the language applicable for real world usage and not only theoretically [59]. As the method of this thesis must be applicable and tested, an editor and EPL statement generator are developed which enables the automatic execution of Step 3 of the method in Chapter 3. For that, a Model Driven Development (MDD) method is utilized in combination with the Xtext[1] framework, which can generate a text editor for the eclipse[2] platform by defining a meta-model and a grammar. The meta-model has to be given as EMF[3] Ecore[4] model, which is a dialect of UML. Thus, the meta-model presented in Figure 5.1 is modeled as Ecore model and integrated into the DSL Xtext project. The grammar has to be defined in the Xtext Grammar Language[5], which is similar to EBNF, but can also define in-memory objects

[1] https://www.eclipse.org/Xtext/
[2] https://www.eclipse.org/
[3] https://www.eclipse.org/modeling/emf/
[4] http://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html
[5] https://www.eclipse.org/Xtext/documentation/301_grammarlanguage.html

graphs for grammar elements. Hence, the EBNF grammar of Listing A.1 is translated into the Xtext grammar language and enhanced with in-memory object assignments to enable automatic processing of the grammar. Consequently, the Xtext framework is utilized to implement a generator for Esper[6] EPL statements (Esper EPL is used because the framework prototype implementation in Chapter 6 utilizes the Esper CEP engine). Table 5.1 shows the mapping between the meta-model elements and according Esper EPL statements. Remaining meta-model elements that are not shown in Table 5.1, do not have a concrete mapping into EPL statements and are only used in the defined constraint language.

[6] http://www.espertech.com/esper/

ConstraintTemplate `pattern StaticWorkload:`	→	Pattern mentioned in EPL Statement Annotation `@Name('StaticWorkload Violation')select ...`
Context `context vmc=VMComponent`	→	EPL from clause `from VMComponent vmc`
EventCollection `VMComponent`	→	EPL event type reference, see Context mapping `VMComponent`
Event `VMComponent(...)`	→	EPL event type `VMComponent(...)`
Parameter `name string`	→	EPL event type property `name string`

EventDefinition
`...`
`def: AVG(value double):`
`avg(c.cpu);`
	→	schema creation, window definition, insert clause
		`create schema AVG(value double);`
		`create window AVGWin.win:length(1) as AVG;`
		`insert into AVG select avg(c.cpu) as value...`

EventSequence `a -> b -> c`	→	EPL event pattern expression `pattern [every a=VMC -> b=AVG -> c=AVG];`
SequenceVariable `... -> a -> ...`	→	EPL event name tags `... -> a=VMComponent -> ...`
Violation / Compliance `violation when: ...`	→	EPL select clause with name annotation `@Name('Watchdog Violation')select * from ...`
Expression (for *Constraint*) `...: a.cpu > 50.0`	→	EPL having clause `... having a.cpu > 50.0`
Expression (for *EventDefinition*) `...: a.cpu > 50.0`	→	EPL select clause `... select a.cpu > 50.0`
EventConstraint and *Expression* `...(a.cpu > 50.0)`	→	EPL where clause (short notation) `...(a.cpu > 50.0)`
WindowConstraint `...last(5)`	→	EPL data window view `...win:length(5)`
EventFilter `...!id`	→	EPL data unique view `...std:unique(id)`
TimeConstraint `...time 5 min`	→	EPL data time view `...win:time(5 min)`

Table 5.1: Table representing the mapping between meta-model elements and Esper EPL statements

6 Dynamic Consistency Checking Framework

The framework for dynamic consistency checking is presented in this chapter. Correspondingly, Section 6.1 presents the design of the framework and according components, while Section 6.2 explains for all design components, how the framework and examples are implemented.

6.1 Design

The Cloud Computing Pattern Consistency Checking (CCPCC) framework is designed to check the consistency status of cloud computing patterns in applications at runtime. Hereby, the framework considers every pattern to be initially consistent, i.e. as soon as the monitored application exposes a behavior that triggers a pattern violation, the pattern is indicated as inconsistent. Main functionalities of the framework are the integration and handling of pattern constraints (enabling Step 4 of the method in Chapter 3), instantiation and controlling of application monitors (enabling Step 5 of the method), and dynamic checking of the consistency status of applications, regarding pattern constraints (the final result of the method). A UML component diagram of the design of the framework and associated components is shown in Figure 6.1. The following color coding is applied on

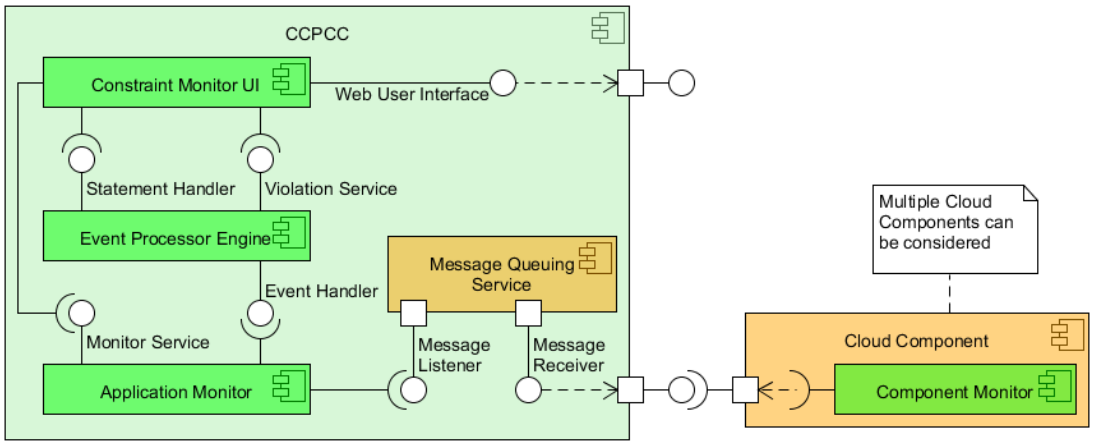

Figure 6.1: UML Component Diagram of the Framework

the diagram: green colored components are self-designed and self to be implemented components, while orange colored components are external services and cloud resources.

The CCPCC framework consists of three main components *Constraint Monitor UI*, the *Event Processor Engine*, *Application Monitor*, and the *Message Queuing Service*. The *Constraint Monitor UI* provides a graphical web user interface that enables user access to the functionalities of the framework. The component depends on a *Statement Handler*, a *Violation Service*, and a *Monitor Service* interface. An *Event Processor Engine* component is responsible for monitoring event streams and applying EPL statements, that enable the indication for violations of pattern consistency. Therefore, the component provides the interfaces *Statement Handler*, *Violation Service*, and *Event Handler*. The *Statement Handler* interface allows to setup the event processing engine to analyze the event stream according to integrated EPL statements. With the *Event Handler* interface, it is possible to inject events into the event stream and declare new event types. Eventually, the *Application Monitor* component enables to control the monitoring of cloud applications by providing a *Monitor Service* interface. By accessing the *Event Handler* interface of the *Event Processor Engine* component, monitors can integrate data in the form of events into the event stream. The communication between the CCPCC framework and monitored applications can be implemented by utilizing a *Message Queuing Service* component, which provides message queues with First In - First Out (FIFO) procedures. FIFO is necessary as the order of events is important for the outcome of defined pattern constraints. The *Application Monitor* component accesses the *Message Listener* interface of the *Message Queuing Service* to integrate runtime information of monitored applications. To stay independent of cloud providers and proprietary software, the monitoring of cloud applications is organized similar to the monitoring methods of Shao et al. [55]. I.e. *Component Monitor* applications are deployed on *Cloud Components* in the environment of a monitored application. *Component Monitors* can be implemented, e.g. as scripts on virtual machines, as code instrumentation in applications, or as applications that are deployed on PaaS offerings to monitor other cloud components. This enables to extract runtime information of monitored cloud components and integrating it via a *Message Receiver* interface of the *Message Queuing Service*.

6.2 Implementation

The framework is implemented as Java web application using the Spring[1] framework for backend implementation and Angular[2] 5 for frontend implementation. Additionally, the framework is organized as Apache Maven Project[3] to provide easy access to the interfaces of the main framework components.

[1] https://spring.io/
[2] https://angular.io/
[3] https://maven.apache.org/

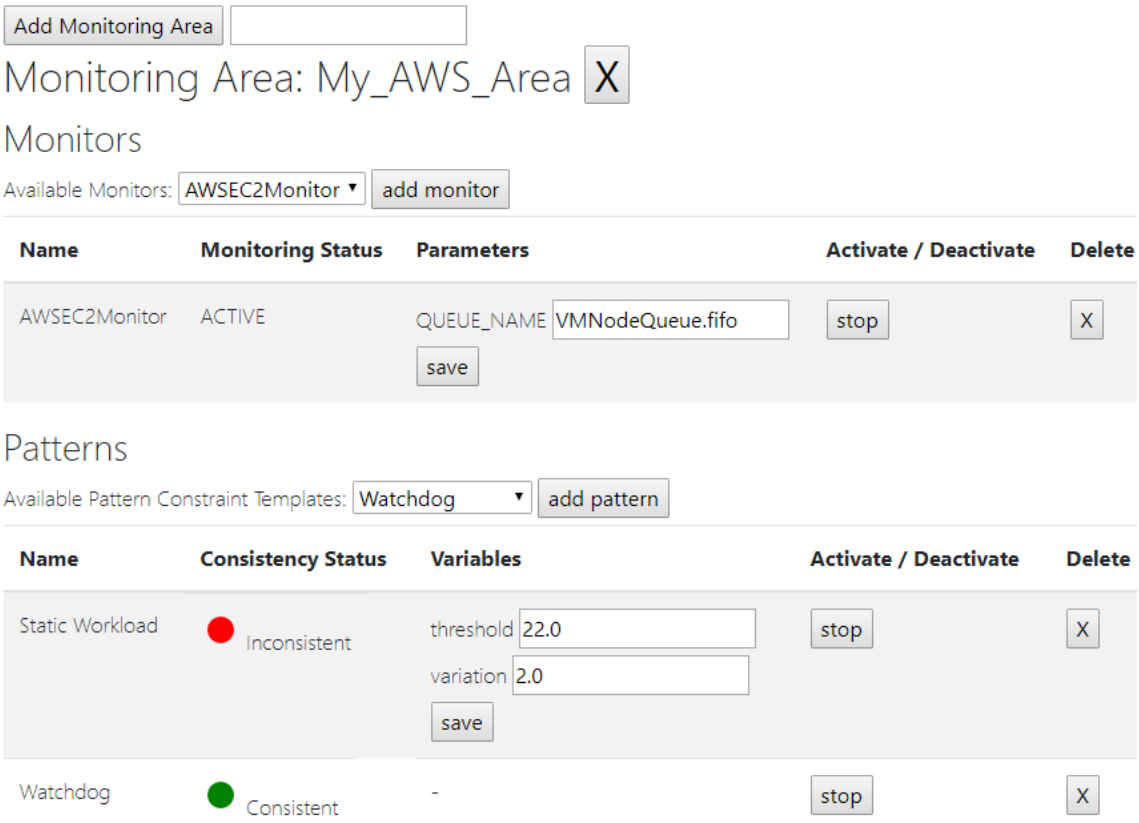

Figure 6.2: Screenshot of the Pattern Monitor View - Web User Interface, showing a monitoring area "My_AWS_Area" which contains one "AWSEC2Monitor" monitor and the two patterns *Static Workload* and *Watchdog*

6.2.1 Constraint Monitor UI

The "Pattern Monitor" view of the web user interface, as shown in Figure 6.2, allows the user to control the consistency checking of applications by organizing component monitors and pattern constraint templates in monitoring areas. All component monitors which are implemented in the *Application Monitor* component, are shown in the dropdown element of available monitors. By selecting one of the available monitors and clicking the "add monitor" button, it is instantiated for the monitoring area and can be configured by setting values for parameters, e.g. the queue name. The instantiated monitors can be activated, deactivated, edited, as well as removed from the monitoring area. Besides, every monitoring area can instantiate pattern constraint templates by selecting a pattern name that is shown in the dropdown element of "Available Pattern Constraint Templates" and clicking the "add pattern" button. As pattern constraint templates allow the usage of constraint variables, every template instance can be configured by setting the values for variables in the list of instantiated templates. Every instantiated pattern constraint template can be activated, deactivated, edited, and removed from the monitoring area. Additionally, they have a consistency status indicator which shows whether the application is either consistent (green circle) or inconsistent (red circle) regarding the considered pattern. If

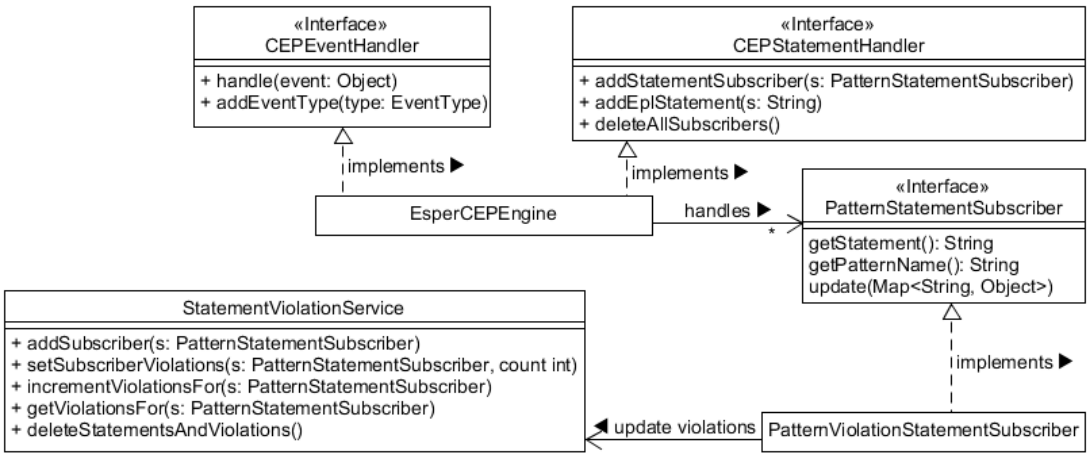

Figure 6.3: UML Class Diagram of the Event Processor Engine Component

the pattern is not active the consistency status is set to undefined with a dash symbol. Furthermore, it is possible to check the violation message by clicking on the red circle, in case the pattern is activated and inconsistent. The violation message shows all properties of event types that are considered in the violation constraint statement.

All entities, i.e. monitoring areas, component monitors and instances, pattern constraint templates and instances, are persisted in a database. For the user it is possible to manually integrate new pattern constraint templates in the form of EPL statements into the application at runtime. This can be done via adding a new instance of the pattern entity in the entity view of the application (entity view not shown because of brevity). However, new component monitors have to be implemented by developers and cannot be added via the user interface at runtime (see Section 6.2.3 for more information).

6.2.2 Event Processor Engine

The *Event Processor Engine* enables the actual analysis of the consistency status of cloud computing patterns. That means a CEP engine is setup with EPL statements, to scan an event stream for low-level and complex events. In the implementation the Esper[4] software is utilized and, thus, the according Esper EPL is used for the engine setup.

Figure 6.3 shows the UML class diagram of the implementation for the *Event Processor Engine*. The provided component interface *Event Handler* is realized by the interface class *CEPEventHandler*, which contains the methods *handle*, to integrate new events into the CEP engine, and the method *addEventType*, which allows to setup the engine to accept new event types. The *CEPStatementHandler* interface class is the realization of the *Statement Handler* provided interface of the component *Event Processor Engine*. It provides methods to add EPL

[4]http://www.espertech.com/esper/

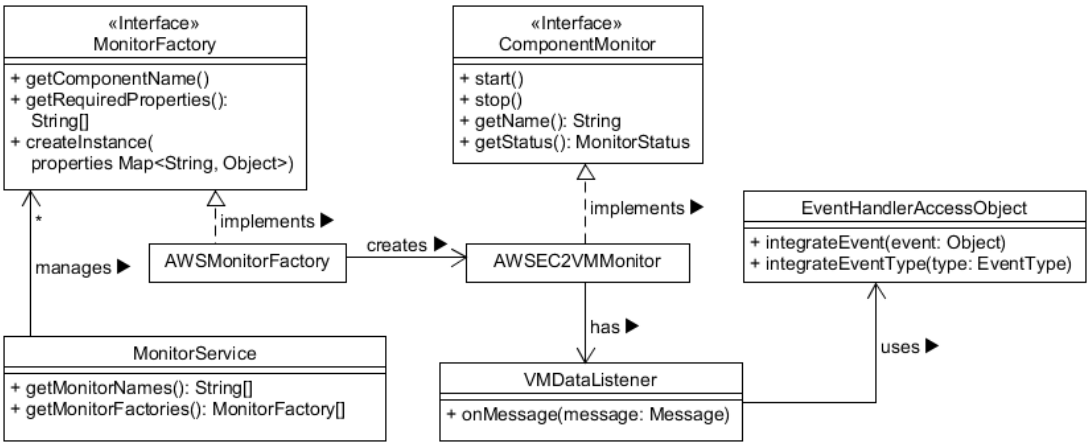

Figure 6.4: UML Class Diagram of the Application Monitor Component

statements (*addEplStatement* method), to define statement subscribers (*addStatementSubscriber* method), and to delete statement subscriptions with the *deleteAllSubscribers* method. Both, the interface classes *CEPEventHandler* and *CEPStatementHandler* are implemented by the class *EsperCEPEngine* to provide a prototype example. The *PatternStatementSubscriber* interface class allows implementing subscriptions on EPL statements which are assigned to a certain pattern. Consequently, the class *PatternViolationStatementSubscriber* implements the interface to indicate that whenever the observed statement is fired, the assigned pattern is violated. Eventually, the *Violation Service* provided interface of the component is implemented by the *StatementViolationService* class. This class can be accessed to retrieve whether violation constraints for patterns have been detected in the event stream of the CEP engine. Furthermore, the class provides methods to add new statement subscribers, to update the violation status of statements and patterns, respectively, and to delete all violations and statements in the service class.

By using interface classes for the component interfaces *Event Handler* and *Statement Handler*, it is possible to exchange the underlying CEP engine with software of other providers. Consequently, when the Esper CEP engine implementation shall be replaced by e.g. Apache Flink[5], it is only necessary to remove the class *EsperCEPEngine* and create one or two, respectively, new classes which implement the interface classes *CEPEventHandler* and *CEPStatementHandler*. No further action is required, as the *Constraint Monitor UI* uses Spring dependency injection on the interfaces.

6.2.3 Application Monitor

The monitoring of cloud applications is implemented in the *Application Monitor* component, which provides the interface *Monitor Service*. Accordingly shown in the UML class diagram

[5] https://flink.apache.org/

in Figure 6.4, the class *MonitorService* is the actual implementation of the provided interface. The class provides methods to get the names of all available monitor implementations in the framework and fetch available monitor factories which enable to create new monitor instances. The interface class *ComponentMonitor* has to be implemented by all specific monitor classes, as given by the example implementation class *AWSEC2VMMonitor*, which is responsible for monitoring virtual machines of the AWS EC2 service[6]. Thus, every monitor must implement functionality that enables the monitor start, stopping, and retrieval of information like name and monitor status. The example implementation for the *Message Queuing Service* component utilizes queues of AWS SQS[7] and implements an event-driven consumer *VMDataListener*, which creates a new event object and integrates it into the CEP engine upon message reception. Therefor, the *EventHandlerAccessObject* class provides methods to integrate events and event types.

To allow the easy integration of new event monitors in the framework, the *MonitorFactory* interface class is given, which has to be implemented for all *ComponentMonitor* implementations. A *MonitorFactory* implementation has to provide information about the component name, required properties and the functionality to create a monitor instance with given properties as key-value pairs. In case new monitors have to be integrated into the framework, it is only necessary to create an implementation for the *MonitorFactory* interface and to create an implementation for the *ComponentMonitor* interface. As the *MonitorService* uses Spring dependency injection, no further actions are required to integrate the monitors manually.

6.2.4 Component Monitor

The framework external component, i.e. the *Component Monitor* is implemented for the prototype implementation as Linux Bash script and, thus, has to be deployed on virtual machine cloud components. It is the counterpart for the *AWSEC2VMMonitor* class which is implemented in the *Application Monitor* component (cf. Section 6.2.3). Thus, the *Component Monitor* prototype extracts virtual machine properties, i.e. the CPU utilization, memory utilization, and a unique identifier. All this information is extracted every second and send to a predefined AWS SQS Queue, by utilizing the AWS CLI[8].

The CPU utilization is extracted by parsing the output of the "top" command, which is pre-installed on standard Ubuntu 16.04 operating systems and provides a dynamic overview of system processes and the CPU utilization. Memory utilization is extracted by parsing the output of the "free" command, which displays the available memory in the Linux machine and is also pre-installed for Ubuntu 16.04. The unique identifier is extracted by using the AWS network, which allows fetching instance metadata and user data [9].

[6] Elastic Compute Cloud - https://aws.amazon.com/de/ec2/
[7] Simple Queue Service - https://aws.amazon.com/de/sqs/
[8] AWS Command Line Interface (CLI) - https://aws.amazon.com/de/documentation/cli/
[9] AWS Metadata and User Data - https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-metadata.html

7 Validation of the Method: Case Studies

This chapter describes how the method defined in Chapter 3 can be applied on selected patterns by the means of constraint examples and by utilizing the pattern analysis, constraint language, and the framework. The method testing for the first pattern is explained in detail to provide a clear understanding on how every step of the method is executed, while the following patterns are addressed concisely, focusing on formal constraint definitions, to maintain brevity.

7.1 Test Setup

The correct functionality to dynamically check the consistency of the patterns is tested in the AWS cloud computing environment and, therefore, the pattern constraints of Chapter 4 are used as the foundation for the test cases. The test environment utilizes the cloud computing service EC2 of the Amazon Web Services. Test data can be generated by using a workload generator that produces workload on the virtual machine, where it is deployed on. The workload generator uses the Java application explained in [11], which provides a simple interface to define the exact amount of workload in the form of CPU utilization that should be experienced on the hosting machine. The tests use EC2 instances with Ubuntu 16.04 as the operating system, pre-installed JRE[1], pre-installed and configured AWS CLI[2], and the AWS network to fetch instance metadata and user data [3]. All this conditions can easily be provided in the form of AMIs[4] and are accordingly used in the test environment.

7.2 Static Workload

In this section, the method is executed for the pattern *Static Workload*. First, the design time phase is elaborated by means of a concrete constraint example and, afterwards, phase two of the method is explained by utilizing the consistency checking framework.

[1] Java Runtime Environment - http://www.oracle.com/technetwork/java/javase/overview/index.html
[2] AWS Command Line Interface (CLI) - https://aws.amazon.com/de/documentation/cli/
[3] AWS Metadata and User Data - https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-metadata.html
[4] Amazon Machine Images (AMI) - https://docs.aws.amazon.com/de_de/AWSEC2/latest/UserGuide/AMIs.html

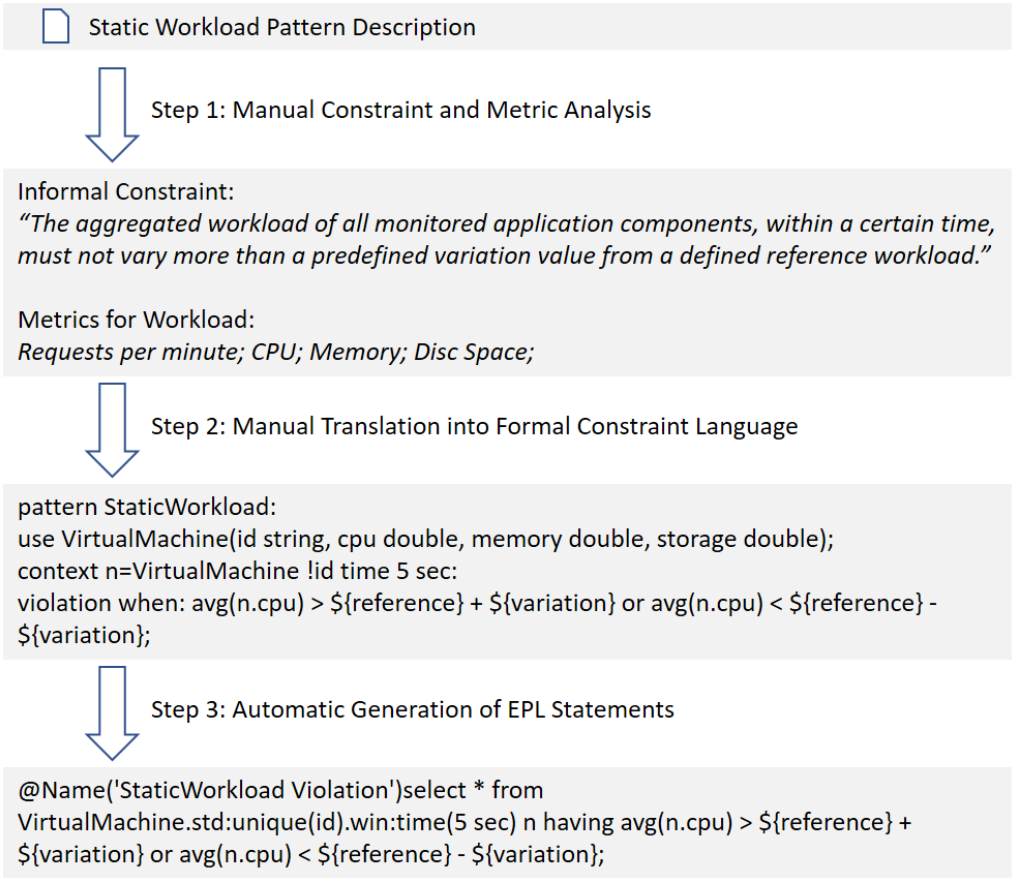

Figure 7.1: Method Execution for Static Workload - Phase 1

7.2.1 Design Time Constraint Definition

The execution of the design time constraint definition phase of the method is depicted in Figure 7.1 for the pattern *Static Workload*. The first step of the method is the analysis of the pattern for constraint statements and for metrics. Hence, we use the constraint definition of the *Static Workload* pattern, as defined in Definition 4.3.2 on page 44, and the according metrics, as defined in Section 4.2.1. The result of the first step is an informal constraint in prose text and measurable metrics for the workload property.

Consequently, the informal constraint and metrics have to be manually translated into a formal constraint template, by utilizing the constraint language of Chapter 5. It is important to remove ambiguities of the constraint by clearly specifying the structure and semantics of the pattern. Hence, for the formal constraint we define the monitored application components as virtual machines, which are formally expressed by the component event *VirtualMachine*. Now, workload metrics for virtual machines are translated as properties for the *VirtualMachine* structure, i.e. the considered properties are CPU utilization (*cpu* with data type "double"), memory utilization (*memory* with data type "double"), disc space utilization (*storage* with data type "double"), and additionally a unique instance identifier

(*id* with data type "string"). The defined virtual machine structure can now be used as context by semantic constraints, thus, the violation statement collects *VirtualMachine* events under the event collection name *n*. As the informal constraint definition suggests the usage of aggregation over a certain time, the event collection *n* only collects the latest virtual machine event for every instance, by defining an event collection filter that filters for the property *id*. Furthermore, a time constraint is added to the context, to only consider events which are not older than 5 seconds, which removes old virtual machine data of instances that already terminated. The semantic violation constraint describes, that whenever the average workload of collected virtual machine instances, in the form of CPU utilization, is varying more than an allowed variation value from a reference value, the pattern *Static Workload* is inconsistent. As the reference value and variation can be different for applications, they are expressed as the constraint variables *${reference}* and *${variation}* which can be set individually for every application at runtime.

After step 2, the constraint is formal and machine readable, such that it can be processed automatically. Hence, for step 3, the language tooling is utilized to automatically translate the constraint template into Esper EPL statements, according to the mapping definition in Table 5.1 on page 70. The generated Esper EPL statement, hence, contains a select-clause with an annotation that indicates violation of the pattern *Static Workload*. In the from-clause is the schema *VirtualMachine* used and combined with a data unique view, considering the EPL type property *id*, and data time view restricting the data window to events not older than five seconds. Expressions of violation statements are mapped to a having-clause while adopting the initial expression as content. The component event definition is not mapped by the generator, as this is equivalent to the creation of CEP event types, which is done by the framework when integrating new monitors. Finishing step 3 completes the first phase of the method, i.e. the Design Time Constraint Definition Phase.

7.2.2 Runtime Consistency Checking

The execution of the second phase of the method is depicted in Figure 7.2. During this phase, the monitored application and the framework implementation, according to Section 6.2, have to be running, such that the web user interface and monitors can be used. To apply step 4 of the method, we have to create a new pattern entity via the web user interface and provide the pattern name, the constraint in the form of EPL statements and, optionally, a pattern description. Afterwards, we can instantiate a pattern for a certain application and monitoring area, respectively, in the "Pattern Monitor" view of the framework. The last step of the method requires to set the constraint variables of the pattern to a concrete value. For this example, we set the value of the "reference" variable to 22.0, indicating that we expect the workload of the monitored virtual machines to experience a CPU utilization of 22%. Furthermore, we define a variation value of 2% by setting the variable "variation" to the value 2.0. Step 5 is completed, as soon as the monitors and the patterns are activated, which results in a dynamic consistency checking of the pattern *Static Workload*.

The pattern is tested by instantiating two AMIs with the defined setup. For the pattern *Static Workload*, the workload generator is configured to produce a CPU utilization of

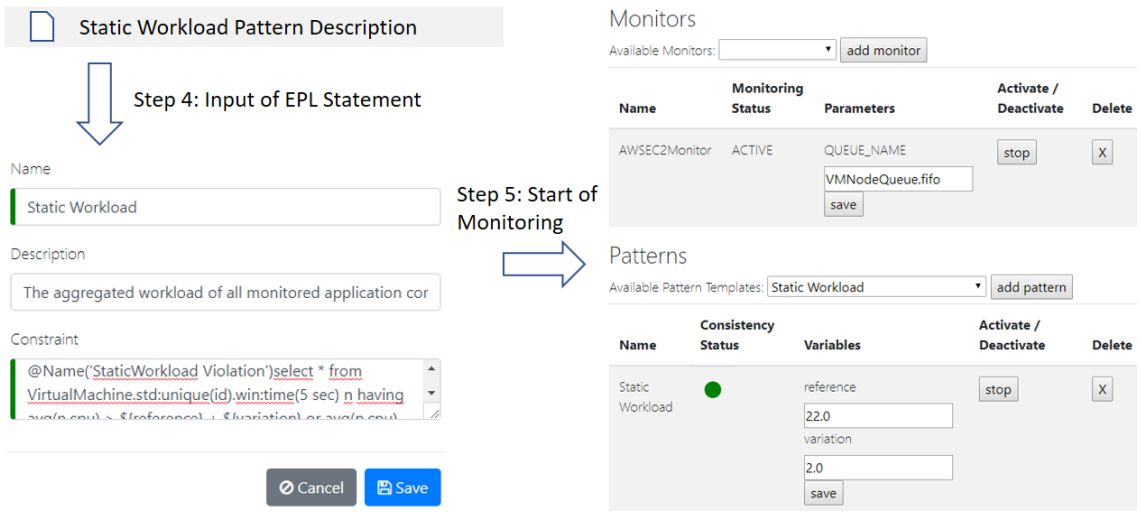

Figure 7.2: Method Execution for Static Workload - Phase 2

22% to simulate workload on the virtual machines. The first test checks whether the pattern is actually consistent as long as the workload on the monitored components is not changed and, thus, it is expected that the framework indicates pattern consistency during the whole test time. The second test aims to check the detection of inconsistencies between the pattern and the monitored application. That means, during test time, the workload generator on one virtual machine is set to produce a workload of 80% instead of 22%, which should result in an average CPU utilization that is varying more than the predefined variation value of 2% and, thus, should trigger violation. Both test executions state that the expected outcomes of the test cases were matching the actual outcomes and, thus, the tests are satisfied for the pattern *Static Workload*.

7.3 Continuously Changing Workload

The second case study is executed for the pattern *Continuously Changing Workload*. During design time of the method we focus on the constraint definition in the formally defined constraint language of Chapter 5. The constraint template is translated from the second constraint statement of the pattern *Continuously Changing Workload* in Definition 4.3.5 on page 46, without considering scaling.

7.3.1 Design Time Constraint Definition

The example in Listing 7.1 shows a constraint template for the pattern *Continuously Changing Workload*, focusing on workload growth. The structure of virtual machine components is defined by the usage of the *VirtualMachine* component event, which has the properties CPU utilization (*cpu*), memory utilization (*memory*), storage utilization (*storage*), and a unique instance identifier (*id*). The interim structure *AverageCpuEvent* is defined

Listing 7.1 Continuously Changing (Growing) Workload Example

```
pattern GrowingWorkload:

use VirtualMachine(id string, cpu double, memory double, storage
    double);

context ne=VirtualMachine !id time 5 sec:
def: AverageCpuEvent(value double): avg(ne.cpu);

context fe=AverageCpuEvent, se=AverageCpuEvent(fe.value < se.value):
violation when sequence: fe -> not se time ${growthTime} min;
```

with an event definition statement. The context contains virtual machine structures and the definition assigns the average CPU utilization of every uniquely identified virtual machine to a parameter with name *value* having type *double*. To only consider the latest running virtual machine instances, a time constraint is added to the context of *AverageCpuEvent*, which removes events that are older than five seconds from the event collection. Finally, a violation constraint statement is defined with the usage of an event sequence which indicates violation, in case the average CPU utilization does not increase within a certain time. The time is kept variable by the definition of the constraint variable *${growthTime}*. That means, for every average CPU utilization measurement, there has to be a following average CPU utilization measurement within a certain amount of minutes, where the value is greater than for the previous event, otherwise the pattern is violated.

7.3.2 Runtime Consistency Checking

For the pattern *Continuously Changing Workload*, considering workload growth, the test setup is equivalent to the setup for *Static Workload*. However, to check consistency of the pattern, the workload is manually changed every minute on one virtual machine by 10 %, such that the average workload grows every minute. Accordingly, in the framework, the constraint variable *${growthTime}* is set to the integer value 2, to indicate that the workload has to grow at least every two minutes. To adhere to the constraint definition of the pattern *Continuously Changing Workload*, the framework must indicate that the pattern is continuously consistent during the test duration. The second test case should check an inconsistent implementation of the pattern, i.e. during test time, the workload is not changed every minute which should violate the constraint for the pattern at least after two minutes of checking, as workload growth is expected. Both test executions stated that the actual outcomes of the test cases were matching the expected outcomes.

7.4 Elasticity Manager

This case study tests the method by the means of the *Elasticity Manager* pattern, which only considers upper case thresholds and scale out operations. Lower thresholds and scale-in operations can be added equivalently and, thus, are not shown in this constraint for brevity.

7.4.1 Design Time Constraint Definition

Listing 7.2 Elasticity Manager Example (Only Considering Upper Threshold)

```
pattern ElasticityManager:

use VirtualMachine(id string, cpu double, memory double, storage
    double);

context n=VirtualMachine !id time 5 sec:
def: AverageCPU(value double): avg(n.cpu);

context vme=VirtualMachine !id time 5 sec:
def: VMCount(number int): count;

context fc=VMCount, sc=VMCount(fc.number < sc.number):
def: IncreaseCountEvent(number int): sc.number;
of sequence: fc -> sc;

context
exceedThreshold=AverageCPU(exceedThreshold.value >= ${threshold}),
belowThreshold=AverageCPU(belowThreshold.value < ${threshold}),
increase=IncreaseCountEvent:
violation when sequence: exceedThreshold -> not belowThreshold time
    ${x} min -> not increase time ${y} min;
```

Listing 7.2 shows an example constraint template for the pattern *Elasticity Manager*. This constraint template is translated from the first *Elasticity Manager* constraint statement in Definition 4.3.8 on page 47. The application structure is defined by the usage of *VirtualMachine*, representing virtual machine components in the application with the properties *id*, *cpu*, *memory*, and *storage*. Then, an event definition statement is defined to calculate the average CPU utilization of all virtual machines as *AverageCPU* events, which are uniquely identified by the *id* property. To only consider the latest running virtual machine instances, a time constraint is added which removes events that are older than five seconds from the event collection. To detect changes in the amount of running virtual machine instances, a statement is provided to count the current amount of running instances with the *VMCount* event definition statement, which uniquely identifies virtual

machines by *id* and only considers events that are not older than five seconds. The time constraint ensures that terminated virtual machines are no longer considered, at least five seconds after termination when counting the amount of running instances. Furthermore, an *IncreaseCountEvent* is defined by the sequence of two consecutive *VMCount* events, where the first count event (*fc*) has a lower number than the second count event (*sc*), to find out if the elasticity manager component executes scaling operations.

Eventually, a violation constraint is defined which utilizes an event collection with name *exceedThreshold*, which collects *AverageCPU* events that have a value greater or equal to a certain threshold. The *belowThreshold* event collection also collects *AverageCPU* events, however, only the events which are below the threshold. Both, the *exceedThreshold* and *belowThreshold* event collections use the *threshold* constraint variable, which can be set during runtime monitoring. Furthermore, the context of the violation statement collects all *IncreaseCountEvents* to check if scaling operations are executed. The event sequence of the violation statement states that the pattern is inconsistent, when the threshold is exceeded continuously for *${x}* minutes and, following, no scaling operation is executed within a time of *${y}* minutes.

7.4.2 Runtime Consistency Checking

The first test case checks whether the framework detects the inconsistency of the pattern *Elasticity Manager* in case the average CPU workload crosses the upper threshold and maintains the high workload for five minutes, but no scaling operation is executed. The constraint variables *${x}* and *${y}* are both set to the value 5, to indicate a time constraint of five minutes in the violation statement. First two initial instances of the AMI are launched and configured to experience a workload of 30% CPU utilization. To filter out false positive test results, the framework monitors the instances over a time of six minutes without changing the workload, which should result in a constant consistent state of the pattern *Elasticity Manager* during that time. After six minutes, the workload on both instances is increased to 80% CPU utilization via the workload generator. As the two instances are not members of an auto scaling group, which would be configured to scale up in case the workload threshold is exceeded, no scaling operations are executed and, thus, the *Elasticity Manager* pattern is not implemented correctly. Hence, after 10 minutes past the incrementation of workload on both instances, the pattern should be indicated as inconsistent. The actual test result matches the expected test result, as exactly after 10 minutes past the workload incrementation, the framework indicates that the pattern *Elasticity Manager* has an inconsistent state.

A further test checks whether the framework states that the pattern is consistent, in case the *Elasticity Manager* is implemented correctly. Thus, an auto scaling group is created for the test AMIs and is configured with a start number of two instances and a maximum number of three instances. A scaling policy is configured which keeps track of the average CPU utilization of the group members and executes scaling operations, whenever the average CPU utilization is greater than 50% for at least five minutes. The scale-in option is disabled for this test, as we only consider upper threshold exceeding and scale-out

operations in the constraint template. At the beginning of the test, both initial instances experience a generated workload of 30% CPU utilization, which is not changed for six minutes. Afterwards, the workload generator is configured on both running instances to produce 80% of CPU utilization. As the instances are members of the configured auto scaling group, a new instance should be provisioned after five minutes of constant average CPU utilization above 50%. The outcome of the test case matches the expected outcome, as the framework indicates that the pattern is still consistent after 10 minutes past the workload increase. The monitoring is continued for further 10 minutes to check whether the pattern stays consistent for three instances, which was the actual outcome and, thus, the test case expectations were met for the pattern *Elasticity Manager*.

7.5 Watchdog

The last case study was conducted for the pattern *Watchdog*. Therefor, we considered the first constraint statement of the pattern, as expressed in Definition 4.3.14 on page 50.

7.5.1 Design Time Constraint Definition

Listing 7.3 Watchdog Example

```
pattern Watchdog:

use VirtualMachine(id string, cpu double, memory double, storage
    double);

context vme=VirtualMachine !id time 5 sec:
def: VMCount(number int): count;

context fe=VMCount, se=VMCount(fe.number > se.number):
def: DecreaseCountEvent(number int): se.number;
of sequence: fe -> se;

context fee=VMCount, see=VMCount(fee.number < see.number):
def: IncreaseCountEvent(number int): see.number;
of sequence: fee -> see;

context re=DecreaseCountEvent, ie=IncreaseCountEvent(re.number <
    ie.number):
violation when sequence: re -> not ie time ${x} min;
```

Listing 7.3 shows an example constraint template for the pattern *Watchdog*. The application structure is defined by the usage of *VirtualMachine*, representing virtual machine components in the application with the properties *id*, *cpu*, *memory*, and *storage*. Afterwards,

a *VMCount* event is defined by counting the amount of uniquely identified virtual machines with a precision of five seconds. I.e., the definition counts the virtual machines that emitted an event which is not older than five seconds to filter out terminated virtual machine instances. The current amount of events is utilized in a further event definition statement, which emits an event *DecreaseCountEvent* every time when there is a sequence of count events where the first event (*fe*) has a higher number compared to a following count event (*se*). That means, in case the amount of running virtual machines is decreased, a *DecreaseCountEvent* is emitted. Vice versa, an *IncreaseCountEvent* is emitted, whenever the amount of running virtual machines is increased. Finally, a violation statement is defined, which indicates that the pattern *Watchdog* is inconsistent, when a *DecreaseCountEvent* is not followed by an *IncreaseCountEvent* within five minutes. The constraint hereby ensures that the watchdog component replaces failed instances within *${x}* minutes.

7.5.2 Runtime Consistency Checking

The constraint variable *${x}* is set to the value 5 for the runtime tests, to indicate a time value of five minutes in the violation statement. The first test case checks whether a correct implementation of the *Watchdog* pattern is equally reflected as consistent in the framework. For this test an auto scaling group is set up to maintain exactly three instances of the predefined AMI. Thus, at the beginning of the test, three instances are running and the framework starts to monitor the application for the pattern *Watchdog*. After one minute of monitoring one instance is manually terminated via the EC2 instance user interface. It is expected that the auto scaling group detects this termination within one minute and, then, starts a new instance of the AMI. The actual outcome for this test case corresponds to the expected outcome, as after less than one minute after the instance termination a new instance was started, and after another minute the instance completed the boot process and was recognized by the framework. The next two test cases check the same proceeding of the first test case, however, two and three, respectively, instances are terminated at the same time. All consistency tests were resulting correspondingly to the expected outcome.

To test the occurrence of the inconsistency of the pattern implementation, the next test case does not set up an auto scaling group according to the *Watchdog* pattern. That means, three instances, which are not a member of an auto scaling group, are initially started and, then, one instance is terminated and not restarted again. As the pattern constraint definition expects the pattern implementation to recover failed instances after at least five minutes, the expected test outcome is the indication of inconsistency in the framework for the pattern *Watchdog* after five minutes, past the termination of an instance. A further test checks if the termination of two instances, followed by the recovery of only one instance results in inconsistency, as the same amount of terminated instances is expected to be recovered by a correct implementation of the *Watchdog* pattern. The actual test outcomes for both test cases correspond to the expected outcomes, as the framework indicates that the pattern *Watchdog* is inconsistent after five minutes, past the manual termination of the instances.

8 Conclusion

The consistency of cloud computing patterns in cloud applications is important to keep the system faithful to the architecture design. Various approaches have been developed to check the consistency and compliance of architecture design and design patterns, respectively. However, existing approaches mostly focus on the analysis of homogeneous and monolithic systems, which are implemented regarding object-oriented design patterns. These approaches are not suitable for the heterogeneous environment of cloud computing applications. Furthermore, so far there is no suitable formal notation for dynamic aspects of cloud computing patterns, that is eligible being applied in combination with an online analysis, to check software architecture compliance and pattern consistency, respectively.

Therefore, this thesis presents a method to achieve dynamic consistency checking of cloud computing patterns. The method is divided into two phases, first, the Design Time Constraint Definition phase and, second, the Runtime Consistency Checking phase. In the first phase, the cloud computing patterns of Fehling et al. [24] are analyzed for metrics and constraints. These constraints and the according metrics are then translated into a formal language and, eventually, mapped into EPL statements. The second phase is initiated by passing the EPL statements as input for a framework, which monitors cloud applications for the provided constraints and updates the pattern consistency status in near real time.

Before applying the method, the cloud computing patterns of Fehling et al. [24] are analyzed for suitability, as there are over 70 patterns defined and not all are suitable for dynamic consistency checking. Nine patterns are selected which are suitable for the method and, consequently, they are further analyzed to specify measurable metrics of cloud components, as well as structural and semantic constraints in informal constraint statements. Regarding the selected patterns, a domain specific constraint language is developed and formally specified by a meta-model, syntax, and semantics. The result is a declarative language which facilitates the definition of structural and semantic constraints, considering runtime information of cloud applications. To support the utilization of the language in real applications, tooling has been developed which enables comfortable constraint declaration in an editor. This also enables the automatic generation of EPL statements for real time processing of the constraints by CEP technology. Consequently, a framework is designed which takes cloud computing pattern constraints, in the form of EPL statements, as input. The framework monitors running cloud applications to extract information that can be processed by a CEP engine. Thus, the given constraints can be checked continuously, to indicate whether the application is consistent regarding the cloud computing patterns or not. To show the feasibility of the framework design, the framework is implemented with an Esper CEP engine and a prototype monitor for virtual machine components.

Eventually, the method is validated on the basis of case studies for the patterns *Static Workload*, *Continuously Changing Workload*, *Elasticity Manager* and *Watchdog*. Concerning this matter, the Amazon Web Services are employed as real world cloud computing environment, for test applications with a simulated workload on virtual machines. The test results state, that the method, utilizing the developed constraint language and framework, actually achieves dynamic consistency checking of cloud computing patterns.

Future Work

Future work can execute the method on the remaining selected cloud computing patterns which were not tested, to validate the method and framework implementation to a greater extent. This would also require the implementation of further component monitors, e.g. for application or load balancer components, which provides more use cases for the framework.

The developed constraint language of this thesis provides all necessary constructs to define constraints for the selected cloud computing patterns. However, the language can serve as a fundamental for further research and, thus, can be improved by specifying more language constructs. For example, additional aggregate functions may be useful for easier constraint expressions, or further filters for event collections might help defining more precise statements. Even though the developed constraint language is mainly designed for the nine selected patterns, it does not necessarily mean that the language is restricted to only these nine patterns. Some patterns were not considered because there are already other approaches (e.g. [41]) that can be adapted to check the consistency of e.g. Management Process patterns. Since the constraint language allows the definition of event sequences, it should be possible to check the consistency of business processes and, thus, Management Process patterns.

As explained by Falkenthal et al. [22], patterns of different abstraction levels can be connected by refinement links. Consequently, in case a specific pattern is monitored, it should be implicitly possible to monitor the corresponding abstract pattern. Hence, future work should conduct further pattern analysis and try to figure out if the method, constraint language, and framework can be applied to more cloud computing patterns.

A Constraint Language Grammar

Listing A.1 Grammar of the constraint language expressed in EBNF notation

```
ConstraintTemplate ::= 'pattern' ID ':' ComponentEvent+ Statement+
ComponentEvent ::= 'use' Event ';'
Event ::= ID '(' Parameter ( ',' Parameter )* ')'
Parameter ::= ID ParameterType
ParameterType ::= 'double' | 'int' | 'string' | 'boolean'
Statement ::= Context ( EventDefinition | Compliance | Violation )
Context ::= 'context' EventCollection ( ',' EventCollection )* ':'
EventCollection ::= ID '=' ID EventConstraint? EventFilter? TimeConstraint?
    WindowConstraint?
WindowConstraint ::= 'last(' ( INT | ConstraintVariable ) ')'
EventConstraint ::= '(' Expression ')'
EventFilter ::= '!' ID
EventDefinition ::= 'def:' Event ':' Expression ( ',' Expression )* ';' ( 'of'
    'sequence:' EventSequence )?
Compliance ::= 'compliance' Constraint
Violation ::= 'violation' Constraint
Constraint ::= 'when' 'sequence:' EventSequence | 'when:' Expression ';'
EventSequence ::= SequenceVariable ( '->' SequenceVariable )* ';'
SequenceVariable ::= 'not'? ID TimeConstraint?
TimeConstraint ::= 'time' ( INT | ConstraintVariable ) TimeUnit
TimeUnit ::= 'milliseconds' | 'sec' | 'min'
Expression ::= OrExpression
OrExpression ::= AndExpression ( 'or' AndExpression )*
AndExpression ::= NotExpression ( 'and' NotExpression )*
NotExpression ::= 'not'? CompareExpression
CompareExpression ::= MathExpression ( CompareOperator MathExpression )*
MathExpression ::= MathTerm ( MathOperator MathTerm )*
MathTerm ::= '(' Expression ')' | Literal | ParameterReference | Function |
    ConstraintVariable
Function ::= AggregateFunction '(' ParameterReference ')' | 'count'
AggregateFunction ::= 'avg' | 'sum' | 'product'
ParameterReference ::= ID '.' ID
Literal ::= STRING | INT | DOUBLE | BOOLEAN
ConstraintVariable ::= '${' ID '}'
ID ::= ('a'..'z'|'A'..'Z') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*
STRING ::= ('a'..'z'|'A'..'Z'|'_'|'0'..'9')+
INT ::= ('0'..'9')+
DOUBLE ::= INT '.' INT
BOOLEAN ::= 'true' | 'false'
CompareOperator ::= '>=' | '<=' | '=' | '!=' | '>' | '<'
MathOperator ::= '+' | '-' | '*' | '/'
```

B Kurzfassung

Cloud-Computing Muster können verwendet werden um Cloud-Anwendungen, basierend auf bewährten und getesteten Lösungen, zu erstellen. Allerdings ist die korrekte Umsetzung dieser Muster nicht über die gesamte Laufzeit der Anwendung garantiert. Inkonsistenzen zwischen der Implementierung und den Cloud-Computing Mustern können zu Abweichungen von Design und Umsetzung führen und letztendlich einen negativen Einfluss auf Qualitätsattribute einer Anwendung haben. Aus diesem Grund beschreibt diese Arbeit eine Methode zur Konsistenzüberprüfung von Cloud-Computing Mustern zur Laufzeit. Neun Cloud-Computing Muster sind ausgewählt und im Detail, auf strukturelle und semantische Bedingungen und Beschränkungen, analysiert. Eine formale Notation wurde entwickelt, welche die Bedingungen und Beschränkungen formal beschreibt und eine automatische Konsistenzüberprüfung dieser Muster ermöglicht. Des Weiteren ist das Design für ein Framework präsentiert, welches Cloud-Anwendungen überwacht und Bedingungen und Beschränkungen von Cloud-Computing Mustern zur Laufzeit prüft. Um die Umsetzbarkeit dieser Methode zu zeigen wurde das Framework implementiert und erfolgreich in einer Cloud-Computing Umgebung, mit simulierter Arbeitsbelastung auf virtuellen Maschinen, für die Muster *Static Workload*, *Continuously Changing Workload*, *Elasticity Manager*, und *Watchdog* getestet.

Bibliography

[1] *AWS Cloud Design Patterns*. URL: http://en.clouddesignpattern.org/ (visited on 01/18/2018) (cit. on p. 20).

[2] C. Ackermann, M. Lindvall, R. Cleaveland. "Towards behavioral reflexion models." In: *Software Reliability Engineering, 2009. ISSRE'09. 20th International Symposium on*. IEEE. 2009, pp. 175–184 (cit. on pp. 15, 25).

[3] P. S. Alencar, D. D. Cowan, C. J.P. d. Lucena. "A formal approach to architectural design patterns." In: *International Symposium of Formal Methods Europe*. Springer. 1996, pp. 576–594 (cit. on p. 21).

[4] C. Alexander, S. Ishikawa, M. Silverstein, J. R. i Ramió, M. Jacobson, I. Fiksdahl-King. *A pattern language*. Gustavo Gili, 1977 (cit. on p. 19).

[5] V. R. Basili. *Software modeling and measurement: the Goal/Question/Metric paradigm*. Tech. rep. 1992 (cit. on p. 38).

[6] V. R. Basili, D. M. Weiss. "A methodology for collecting valid software engineering data." In: *IEEE Transactions on software engineering* 6 (1984), pp. 728–738 (cit. on p. 38).

[7] U. Breitenbücher. "Eine musterbasierte Methode zur Automatisierung des Anwendungsmanagements." PhD thesis. Institut für Architektur von Anwendungssystemen, 2016 (cit. on p. 32).

[8] U. Breitenbücher, T. Binz, C. Fehling, O. Kopp, F. Leymann, M. Wieland. "Policy-aware provisioning and management of cloud applications." In: *International Journal On Advances in Security* 7.1&2 (2014), pp. 15–36 (cit. on p. 34).

[9] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, M. Wieland. "Policy-aware provisioning of cloud applications." In: *Conference on Emerging Security Information, Systems and Technologies* (2013) (cit. on p. 34).

[10] *Business process model and notation (BPMN) version 2.0*. URL: http://www.omg.org/spec/BPMN/2.0/ (visited on 01/18/2018) (cit. on p. 37).

[11] Caffinc Blog. *CPU Load Generator in Java*. Mar. 9, 2016. URL: https://caffinc.github.io/2016/03/cpu-load-generator/ (visited on 05/08/2018) (cit. on p. 77).

[12] *Cloud Patterns*. URL: http://cloudpatterns.org (visited on 01/18/2018) (cit. on p. 20).

[13] C. Czepa, H. Tran, U. Zdun, T. T. T. Kim, E. Weiss, C. Ruhsam. "On the Understandability of Semantic Constraints for Behavioral Software Architecture Compliance: A Controlled Experiment." In: *Software Architecture (ICSA), 2017 IEEE International Conference on*. IEEE. 2017, pp. 155–164 (cit. on pp. 15, 25).

[14] L. De Silva, D. Balasubramaniam. “Controlling software architecture erosion: A survey.” In: *Journal of Systems and Software* 85.1 (2012), pp. 132–151 (cit. on p. 16).

[15] M. von Detten, M. Platenius. “Improving Dynamic Design Pattern Detection in Reclipse with Set Objects.” In: *Proceedings of the 7th International Fujaba Days*. 2009 (cit. on p. 25).

[16] B. Di Martino, A. Esposito. “A rule-based procedure for automatic recognition of design patterns in UML diagrams.” In: *Software: Practice and Experience* 46.7 (2016), pp. 983–1007 (cit. on pp. 15, 20, 21, 24, 30).

[17] J. Dietrich, C. Elgar. “A formal description of design patterns using OWL.” In: *Software Engineering Conference, 2005. Proceedings. 2005 Australian*. IEEE. 2005, pp. 243–250 (cit. on pp. 23, 24).

[18] S. K. Doddavula, I. Agrawal, V. Saxena. “Cloud Computing Solution Patterns: Application and Platform Solutions.” In: *Cloud Computing*. Springer, 2013, pp. 221–239 (cit. on p. 15).

[19] A. H. Eden. “A theory of object-oriented design.” In: *Information Systems Frontiers* 4.4 (2002), pp. 379–391 (cit. on p. 23).

[20] A. H. Eden. “Formal specification of object-oriented design.” In: *International Conference on Multidisciplinary Design in Engineering*. 2001, pp. 256–263 (cit. on p. 21).

[21] M. Falkenthal, U. Breitenbücher, F. Leymann. “The Nature of Pattern Languages.” In: *Pursuit of Pattern Languages for Societal Change*. Ed. by P. Baumgartner, Gruber-MueckeTina, R. Sickinger. Berlin: epubli, 2018 (cit. on p. 19).

[22] M. Falkenthal, J. Barzen, U. Breitenbücher, C. Fehling, F. Leymann, A. Hadjakos, F. Hentschel, H. Schulze. “Leveraging Pattern Application via Pattern Refinement.” In: *Proceedings of the International Conference on Pursuit of Pattern Languages for Societal Change (PURPLSOC)*. epubli GmbH, 2016, pp. 38–61 (cit. on pp. 19, 88).

[23] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Cloud Computing Patterns*. 2014. URL: http://cloudcomputingpatterns.org (visited on 01/18/2018) (cit. on pp. 16, 20).

[24] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Cloud computing patterns: fundamentals to design, build, and manage cloud applications*. Springer Science & Business Media, 2014 (cit. on pp. 15–17, 19, 20, 24, 29–31, 33, 35, 40, 87).

[25] R. B. France, D.-K. Kim, S. Ghosh, E. Song. “A UML-based pattern specification technique.” In: *IEEE transactions on Software Engineering* 30.3 (2004), pp. 193–206 (cit. on pp. 21–23, 30).

[26] E. Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995 (cit. on pp. 19, 23).

[27] E. Gamma, R. Johnson, R. Helm, J. Vlissides. *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Pearson Deutschland GmbH, 2011 (cit. on pp. 15, 19, 20, 22, 24).

[28] S. Gilbert, N. Lynch. “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services.” In: *Acm Sigact News* 33.2 (2002), pp. 51–59 (cit. on p. 35).

[29] *Google Cloud Design Patterns*. URL: https://cloud.google.com/apis/design/design_patterns (visited on 02/21/2018) (cit. on p. 20).

[30] M. Hapner, R. Burridge, R. Sharma, J. Fialli, K. Stout. “Java message service.” In: *Sun Microsystems Inc., Santa Clara, CA* (2002), p. 9. URL: http://download.oracle.com/otn-pub/jcp/jms-2_0-pr-spec/JMS20.pdf (visited on 03/31/2018) (cit. on p. 42).

[31] D. Heuzeroth, S. Mandel, W. Lowe. “Generating design pattern detectors from pattern specifications.” In: *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*. IEEE. 2003, pp. 245–248 (cit. on pp. 21, 22).

[32] G. Hohpe, B. Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional, 2004 (cit. on pp. 19, 20).

[33] *ISO/IEC 25010 Standard*. URL: http://iso25000.com/index.php/en/iso-25000-standards/iso-25010 (visited on 03/13/2018) (cit. on p. 40).

[34] K. Képes, U. Breitenbücher, M. P. Fischer, F. Leymann, M. Zimmermann. “Policy-Aware Provisioning Plan Generation for TOSCA-based Applications.” In: *Proceedings of The Eleventh International Conference on Emerging Security Information, Systems and Technologies (SECURWARE 2017), 10-14 September 2017, Rome, Italy*. Xpert Publishing Services, 2017, pp. 142–149 (cit. on p. 34).

[35] D. Kirasić, D. Basch. “Ontology-based design pattern recognition.” In: *International Conference on Knowledge-Based and Intelligent Information and Engineering Systems*. Springer. 2008, pp. 384–393 (cit. on p. 24).

[36] A. Lauder, S. Kent. “Precise visual specification of design patterns.” In: *European Conference on Object-Oriented Programming*. Springer. 1998, pp. 114–134 (cit. on pp. 21–23).

[37] F. Leymann, D. Roller. *Production workflow: concepts and techniques*. Prentice Hall PTR Upper Saddle River, 2000 (cit. on p. 40).

[38] D. Luckham. *The power of events*. Vol. 204. Addison-Wesley Reading, 2002 (cit. on p. 27).

[39] J. Ludewig, H. Lichter. *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken*. dpunkt. verlag, 2013 (cit. on pp. 38, 40).

[40] P. Mell, T. Grance, et al. “The NIST definition of cloud computing.” In: (2011) (cit. on pp. 34, 35).

[41] E. Mulo, U. Zdun, S. Dustdar. “Domain-specific language for event-based compliance monitoring in process-driven SOAs.” In: *Service Oriented Computing and Applications* 7.1 (2013), pp. 59–73 (cit. on pp. 25–27, 31, 37, 88).

[42] E. Mulo, U. Zdun, S. Dustdar. “Monitoring web service event trails for business compliance.” In: *Service-oriented computing and applications (SOCA), 2009 IEEE international conference on*. IEEE. 2009, pp. 1–8 (cit. on p. 27).

[43] G. C. Murphy, D. Notkin, K. Sullivan. “Software reflexion models: Bridging the gap between source and high-level models.” In: *ACM SIGSOFT Software Engineering Notes* 20.4 (1995), pp. 18–28 (cit. on p. 25).

[44] OASIS. “TOSCA Specification.” In: (2013) (cit. on p. 32).

[45] OMG. *OMG Object Constraint Language (OCL), Version 2.4*. Object Management Group, 2014. URL: http://www.omg.org/spec/OCL/2.4/ (visited on 04/16/2018) (cit. on pp. 23, 64).

[46] OMG. *OMG Unified Modelling Language (UML), Version 2.5.1*. Object Management Group, 2017. URL: https://www.omg.org/spec/UML/2.5.1/ (visited on 04/16/2018) (cit. on pp. 23, 67).

[47] OWL Working Group. *Web Ontology Language*. 2004. URL: https://www.w3.org/TR/owl2-overview/ (visited on 04/30/2018) (cit. on p. 24).

[48] G. Palshikar et al. “Simple algorithms for peak detection in time-series.” In: *Proc. 1st Int. Conf. Advanced Data Analysis, Business Analytics and Intelligence*. 2009, pp. 1–13 (cit. on p. 45).

[49] D. E. Perry, A. L. Wolf. “Foundations for the study of software architecture.” In: *ACM SIGSOFT Software engineering notes* 17.4 (1992), pp. 40–52 (cit. on p. 21).

[50] S. Qanbari, S. Pezeshki, R. Raisi, S. Mahdizadeh, R. Rahimzadeh, N. Behinaein, F. Mahmoudi, S. Ayoubzadeh, P. Fazlali, K. Roshani, et al. “IoT design patterns: computational constructs to design, build and engineer edge applications.” In: *Internet-of-Things Design and Implementation (IoTDI), 2016 IEEE First International Conference on*. IEEE. 2016, pp. 277–282 (cit. on p. 20).

[51] L. Reinfurt, U. Breitenbücher, M. Falkenthal, F. Leymann, A. Riegg. “Internet of Things Patterns for Devices: Powering, Operating, and Sensing.” In: *International Journal on Advances in Internet Technology* (2017), pp. 106–123 (cit. on p. 20).

[52] D. Riehle, H. Züllighoven. “Understanding and using patterns in software development.” In: *Tapos* 2.1 (1996), pp. 3–13 (cit. on p. 20).

[53] S. Rozsnyai, R. Vecera, J. Schiefer, A. Schatten. “Event cloud-searching for correlated business events.” In: *E-Commerce Technology and the 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services, 2007. CEC/EEE 2007. The 9th IEEE International Conference on*. IEEE. 2007, pp. 409–420 (cit. on p. 27).

[54] K. Saatkamp, U. Breitenbücher, O. Kopp, F. Leymann. “Topology splitting and matching for multi-cloud deployments.” In: *8th Int. Conf. on Cloud Computing and Service Sciences (CLOSER 2017)*. 2017 (cit. on p. 32).

[55] J. Shao, H. Wei, Q. Wang, H. Mei. “A runtime model based monitoring approach for cloud.” In: *Cloud Computing (CLOUD), 2010 IEEE 3rd international conference on*. IEEE. 2010, pp. 313–320 (cit. on pp. 26, 27, 31, 32, 72).

[56] J. M. Smith, D. Stotts. “Elemental Design Patterns: A formal semantics for composition of OO software architecture.” In: *Software Engineering Workshop, 2002. Proceedings. 27th Annual NASA Goddard/IEEE*. IEEE. 2002, pp. 183–190 (cit. on pp. 16, 21, 22).

[57] T. Taibi, D. C. L. Ngo. “Formal Specification of Design Patterns - A Balanced Approach.” In: *Journal of Object Technology* 2.4 (2003), pp. 127–140 (cit. on pp. 22, 24).

[58] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, S. T. Halkidis. “Design pattern detection using similarity scoring.” In: *IEEE transactions on software engineering* 32.11 (2006) (cit. on p. 15).

[59] M. Völter. “Best practices for DSLs and model-driven development.” In: *Journal of Object Technology* 8.6 (2009), pp. 79–102. URL: http://www.jot.fm/issues/issue_2009_09/column6/ (cit. on pp. 31, 64, 68).

[60] D. H. Vu et al. “Specifying object-oriented design patterns using OWL.” In: (2010) (cit. on pp. 21, 24).

[61] L. Wendehals. “Improving design pattern instance recognition by dynamic analysis.” In: *Proc. of the ICSE 2003 Workshop on Dynamic Analysis (WODA), Portland, USA*. 2003, pp. 29–32 (cit. on pp. 25, 26).

[62] L. Wendehals, A. Orso. “Recognizing behavioral patterns at runtime using finite automata.” In: *Proceedings of the 2006 international workshop on Dynamic systems analysis*. ACM. 2006, pp. 33–40 (cit. on p. 26).

[63] *Windows Azure Application Patterns*. URL: https://blogs.msdn.microsoft.com/jmeier/2010/09/11/windows-azure-application-patterns/ (visited on 02/21/2018) (cit. on p. 20).

All links were last followed on July 2, 2018.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature