Monitoring Behavioral Compliance with Architectural Patterns based on Complex Event Processing

Christoph Krieger[1], Uwe Breitenbücher[1], Michael Falkenthal[1],
Frank Leymann[1], Vladimir Yussupov[1], and Uwe Zdun[2]

[1] Institute of Architecture of Application Systems, University of Stuttgart, Germany
[lastname]@iaas.uni-stuttgart.de
[2] University of Vienna, Faculty of Computer Science, Austria
[firstname.lastname]@univie.ac.at

Abstract. Architectural patterns assist in the process of architectural decision making as they capture architectural aspects of proven solutions. In many cases, the chosen patterns have system-wide implications on non-functional requirements such as availability, performance, and resilience. Ensuring compliance with the selected patterns is of vital importance to avoid architectural drift between the implementation and its desired architecture. Most of the patterns not only capture structural but also significant behavioral architectural aspects that need to be checked. In case all properties of the system are known before runtime, static compliance checks of application code and configuration files might be sufficient. However, in case aspects of the system dynamically evolve, e.g., due to manual reconfiguration, compliance with the architectural patterns also needs to be monitored during runtime. In this paper, we propose to link compliance rules to architectural patterns that specify behavioral aspects of the patterns based on runtime events using stream queries. These queries serve as input for a complex event processing component to automatically monitor architecture compliance of a running system. To validate the practical feasibility, we applied the approach to a set of architectural patterns in the domain of distributed systems and prototypically implemented a compliance monitor.

Keywords: Architecture Compliance · Architectural Patterns · Behavioral Compliance Monitoring · Complex Event Processing.

1 Introduction

While designing complex software systems, various architectural decisions need to be made by software architects and later implemented in code by software developers. Architectural patterns can assist in the process of architectural decision making and documentation, as they capture structural and behavioral architectural aspects of proven solutions that are documented in a generic and technology-independent way [9, 22, 23]. An example of an architectural pattern in the domain of distributed systems is the *Circuit Breaker* pattern [13]. This pattern describes how to avoid cascading failures in case of network or remote services failures by wrapping remote function calls with a proxy that monitors failures and reacts in a similar way as an electrical circuit breaker.

However, the correct realization of the architectural decisions is often not ensured during the entire life-cycle of a software system. Reasons for this are inadequate implementation by application developers, non-compliant deployment of application components, and also operator errors during manual configuration of the running system. As a consequence, the software system drifts apart from the original design specification, which is commonly referred to as *architectural drift* [16]. This is particularly problematic in the case of the chosen architectural patterns as they are concerned with essential aspects of the software architecture that often have system-wide implications on quality aspects such as availability, performance, and resilience. For example, a non-compliant realization of the aforementioned Circuit Breaker pattern can lead to cascading failures of services which harm the reliability of the overall system. Thus, architectural compliance checks are needed to ensure the correct implementation, deployment, and configuration of architectural aspects described by the chosen patterns. In case behavioral aspects of the system dynamically evolve during runtime, architectural compliance can not be guaranteed by simply checking the application code and configuration files during design-time. For example, operator errors during configuration of a running system may cause behavioral deviation from the intended architecture. In such cases, it is of vital importance to not just check architectural compliance during design-time but also monitor the compliance during run-time.

Therefore, the research question of this work is: “How can we automatically monitor a system’s architectural compliance based on behavioral aspects described in architectural patterns?”. To tackle this issue, we present an approach for architectural compliance monitoring based on complex event processing. We propose to specify behavioral aspects described by architectural patterns as so-called *Pattern Compliance Rules* that serve as input for an *architectural compliance monitoring system* to monitor the compliance of an application with the specified patterns. Thereby, we show how behavioral compliance aspects described in architectural patterns can be specified based on events using stream query languages and how the runtime events can be automatically monitored while the system executes. Moreover, to validate the practical feasibility of the presented approach, we applied the concept to a set of architectural patterns for designing distributed systems and prototypically implemented a compliance monitor using Esper.

2 Fundamentals and Motivation

In this section, we describe fundamentals required for understanding this paper. Moreover, we introduce a motivating scenario that is used throughout the paper to explain the presented approach.

2.1 Patterns and Design Decisions

Patterns describe proven solutions for problems that frequently reoccur in a certain context [2]. Patterns are documented in an abstract way and typically follow a well-defined structure comprising a *pattern's name*, a *problem description*, details about the *context* in which they can be applied, and a proven *solution.* In the domain of software architecture, various *pattern languages* exist that describe proven solutions for designing application architectures. For example, the *Circuit Breaker* pattern [13] tackles the problem of cascading failures in distributed systems when networks or remote services fail. Here, function calls to remote services are wrapped with a proxy that monitors failures and reacts similarly as electrical circuit breakers. In case a given threshold of consecutive failures is exceeded, the circuit breaker trips and for a specified timeout period all attempts to invoke the remote service will fail immediately. Another architecture pattern useful for distributed applications is the *Watchdog* pattern [7] that describes how failing application components can be detected and replaced automatically to ensure high availability. Thus, using such patterns significantly helps in the architectural decision making as problems at hand can be solved by using proven solutions. Moreover, as patterns provide developers with information about the rationale and consequences of a solution, they can be used to assist the documentation of *architectural design decisions* (ADD) [21].

2.2 Motivating Scenario

In this section, we introduce a motivating scenario that is used throughout the paper to motivate and explain our approach. When designing microservice-based applications, development teams are faced with the complexity of a distributed system and need to make various design decisions to build fault tolerant services. Patterns provide an effective way to help making and documenting such important design decisions as they capture architectural aspects of proven solutions together with their rationale and consequences. The following are common pattern-based design decisions made in practice to design highly available microservice-based applications (see e.g. [8]):

ADD01: To prevent the application from excessive load, caused by malicious or misconfigured clients, the *Rate Limiting* pattern [18] needs to be implemented that enforces a request limit of four requests per second for each client.

ADD02: To prevent cascading failures, the *Circuit Breaker* pattern [13] needs to be implemented for all remote function calls. A circuit breaker needs to trip in case three consecutive calls of the remote function fail. The specified timeout for an open circuit breaker is five seconds.

ADD03: To ensure sufficient availability, the *Watchdog* pattern [7] should be applied to detect and replace failed application component instances.

The implementation and configuration of the documented architectural patterns is done manually by software developers which is error-prone, meaning that parts of the patterns may be misconceived, accidentally overlooked, or even intentionally ignored due to time pressure. Such configuration and implementation errors can have an implication on the overall resilience and availability of the system. For example, behavioral non-compliance of circuit breakers, e.g., to the specified threshold of consecutive failures defined in the aforementioned ADD, can lead to cascading failures and eventually jeopardize the whole application. Just as critical for ensuring availability of the application is the correct implementation of the Watchdog pattern. A watchdog is often implemented by simply configuring an existing monitoring component. As an example, in case of virtual machines running on Amazon EC2, an auto scaling group can be configured that specifies a minimum number of virtual machine instances in that group. By replacing failed component instances, Amazon's EC2 Auto Scaling ensures that the group never goes below the specified minimum. However, creating such configurations or reconfiguring existing ones can become a complex and error-prone task which often results in non-compliant system behavior [14]. Thus, architecture compliance checks are needed to ensure the correct implementation, deployment, and configuration of a system's architectural aspects [20]. In case all application components, their relationships, and architectural decisions to be implemented are known before runtime of the application, checking compliance using static analysis might be enough. However, in case behavioral aspects of the system can be dynamically reconfigured during runtime the compliance can no longer be guaranteed by static compliance checks alone. For such applications, it is of vital importance to not just check architectural compliance during design-time but also monitor the compliance during run-time. In this paper, we propose to specify behavioral aspect of architectural patterns based on run-time events which can be used to automatically monitor architecture compliance.

2.3 Complex Event Processing

Complex Event Processing (CEP) is a set of techniques and tools for analyzing and controlling complex series of interrelated events, e.g., produced by distributed systems [11]. This technique can be used to monitor behavioral architecture compliance aspects of a system based on runtime events. *Stream query languages*, are used to configure CEP engines to observe live data streams of events and aggregate so-called low-level events into complex (high-level) events to enable discovery of event patterns having semantic significance in a specific context [12]. For example, a typical low-level event is a network event, such as an HTTP request sent by a service or the response to that request. While a single request provides no significant behavioral information, multiple ones observed in particular order and time can provide more insights into the system behavior and help to recognize non-compliant behavior. One frequently used Stream Query Language is Esper Event Processing Language (EPL) which is included as a part of Esper's open source CEP engine [4]. Statements in Esper EPL have an SQL-like syntax containing standard query clauses such as *SELECT*, *FROM*, and *WHERE*. In this context, event streams represent data sources, whereas events serve as the basic unit of data. Moreover, Esper EPL provides multiple event pattern operators and time windows to facilitate querying of event data. For example, Listing 1.1 shows an EPL statement that can be used to analyze network events emitted by a running application to detect HTTP responses that exceed a response time of 1 second. The statement demonstrates the idea of emitting a high-level ResponseTimeout event in case a HttpRequest event is not followed by a corresponding HttpResponse event within a time window of 1 second. For a complete overview of the Esper EPL, we refer to the documentation provided by Esper [4].

```
insert into ResponseTimeout
from pattern [every a=HttpRequest −> not b=HttpResponse(a.sender = b.receiver)
and timer:within(1 sec)];
```

Listing 1.1. An example of an EPL statement for analyzing a stream of HTTP events.

3 An Approach for Monitoring Behavioral Compliance with Architectural Patterns

The main idea of our approach is to introduce so called Pattern Compliance Rules which serve as configuration code for monitoring behavioral compliance with architectural patterns. A Pattern Compliance Rule (PCR) contains a set of stream query language statements that specify behavioral aspects of an architectural pattern based on runtime events which can be automatically monitored while a system executes. The overall concept of our approach is shown in Figure 1. There, Pattern Compliance Rules are managed in a *Pattern Compliance Rule Repository*. Similar to the patterns itself, the PCRs managed in the repository are application-agnostic which has the advantage that existing PCRs can be reused, hence reducing the required effort for the creation of compliance monitoring code. In addition, *Instrumentation Templates* are associated with each Pattern Compliance Rule providing program or configuration code that can be used for the instrumentation of a monitored application to emit the necessary runtime events. Thereby, each Instrumentation Template targets a specific technology, e.g., programming language and instrumentation mechanism. Based on the architectural patterns that should be monitored, a set of corresponding Pattern Compliance Rules are selected from the repository and bound to application-specific details of architectural design decisions. The resulting *application-specific PCRs* serve as configuration code for a complex event processing engine of a specialized software component, called a *Pattern Compliance Monitor*. Moreover, the Instrumentation Templates associated with the chosen PCRs can be used as a basis for the instrumentation of the monitored application to create the necessary runtime events. Each event represents the occurrence of an activity within the monitored application. For example, an event may represent a request sent between application components and contain information about the source and target of the request. While the application

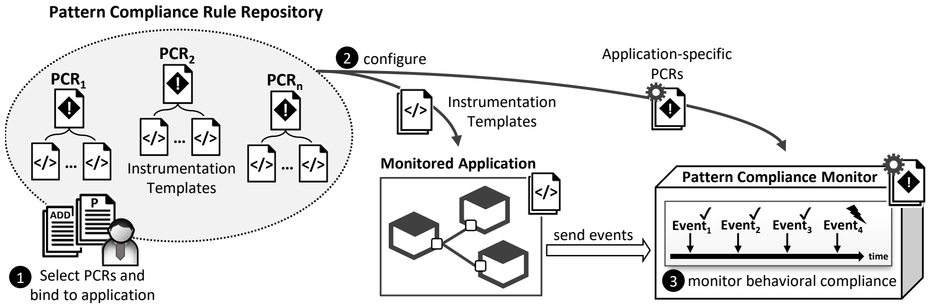
Pattern Compliance Rule Repository
PCR_1
PCR_2
PCR_n
</> … </>
Instrumentation Templates
ADD
P
1 Select PCRs and bind to application
2 configure
Instrumentation Templates
Application-specific PCRs
Monitored Application
send events
Pattern Compliance Monitor
$Event_1$
$Event_2$
$Event_3$
$Event_4$
time
3 monitor behavioral compliance

Fig. 1. Conceptual overview of the approach for monitoring behavioral compliance with architectural patterns.

executes, the runtime events are continuously sent to the Pattern Compliance Monitor. There, the stream of events is observed to monitor compliance with the expected behavior described by the Pattern Compliance Rules.

3.1 Method

Figure 2 depicts a step-wise method for the configuration of the monitoring environment. The method consists of five steps, namely (i) identifying architectural patterns to be implemented in an application, (ii) selecting the corresponding Pattern Compliance Rules from the Pattern Compliance Rule Repository, (iii) optionally creating Pattern Compliance Rules that are not already contained in the repository, (iv) binding the selected PCRs to application-specific details, and (v) using the resulting application-specific PCRs as configuration code for behavioral architecture compliance monitoring and instrument the monitored application to emit the necessary runtime events. In the following, we describe every step in more detail and exemplify the method based on the Rate Limiting pattern described in the motivating scenario.

Step 1: Identify Patterns In the first step, the architectural documentation, which is created during the system design phase, is analysed to identify a set of architectural patterns that need to be realized by the implementation and deployment of the application. For example, in case of the motivating scenario presented in Section 2.2, the three architectural patterns Rate Limiting, Circuit Breaker, and Watchdog can be identified by analysing the documentation of ADD01, ADD02, and ADD03. Other data sources potentially containing pattern descriptions could be design diagrams or other formats of architecture documentation. The resulting set of identified patterns is then passed as an input to the next step.

Steps 2 & 3: Select or Create PCRs In the second step, for each identified pattern a corresponding PCR is chosen from the PCR Repository as shown in

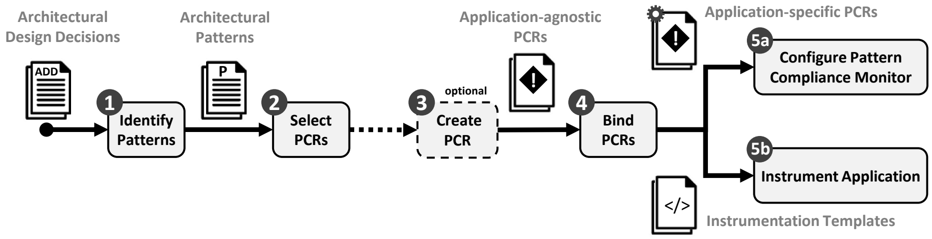
Architectural Design Decisions
ADD
1
Identify Patterns
Architectural Patterns
P
2
Select PCRs
optional
3
Create PCR
Application-agnostic PCRs
4
Bind PCRs
Application-specific PCRs
5a
Configure Pattern Compliance Monitor
5b
Instrument Application
</>
Instrumentation Templates

Fig. 2. An overview of the step-wise method for the configuration of the monitoring environment.

Figure 2. The PCRs managed in the repository are so-called *application-agnostic PCRs*, meaning that instead of application-specific implementation details, they contain placeholders in the form of variables. This has the main advantage that PCRs managed in the repository can be reused, hence reducing the required effort for the creation of monitoring code. Optionally, in case a corresponding PCR for one of the identified architectural patterns does not already exist, a new one is created and added to the repository. For example, in the case of the Rate Limiting pattern, an application-agnostic Rate-Limiting PCR is selected or created that describes the expected behavior of application components implementing this pattern based on runtime events. The pattern states that the number of requests that can be made by a client should be restricted to a defined limit. Listing 1.2 depicts an exemplary Rate-Limiting PCR described using the Esper EPL. There, variables are marked in bold. The PCR describes an event pattern of HTTP requests that violates the behavior described by the Rate Limiting pattern. There, each request made by a client is represented by an *HttpRequest* event containing the *source* the request originates from, e.g. the client's IP address, and the *responseCode* returned by the server. The EPL statement (line 3-7) observes the stream of HttpRequest events per client and selects the aggregation of events as a *RateLimitViolation* in case the number of accepted HTTP requests observed in a given time interval exceeds a predefined limit. The request limit and the time interval for rate-limiting are defined as variables and can be bound to concrete values based on application-specific details.

```
1  create schema HttpRequest(source, responseCode)
2
3  @Name('Rate Limiting Violation')
4  insert into RateLimitViolation
5  select count(*) from HttpRequest#time_batch(interval sec)
6  where responseCode = '200'
7  group by source having count(*) > requestLimit;
```

Listing 1.2. An exemplary application-agnostic PCR for the Rate Limiting pattern defined using the Esper EPL.

Step 4: Bind PCRs In the fourth step, the application-agnostic PCRs are bound to application-specific details to (i) align them with application-specific design decisions and (ii) to serve as executable configuration code for the Pattern Compliance Monitor shown in Figure 2. This means all variables, contained in a rule, are replaced with application-specific data, e.g., documented in architectural design decisions. For example, to align the Rate Limiting PCR with the architectural design decision ADD01 documented in the motivating scenario, the *requestLimit* is defined as four requests and the *interval* is set to one second.

Step 5: Configure Monitoring Environment In step 5, the set of previously created application-specific PCRs are used as configuration code for the Pattern Compliance Monitor as shown in Figure 1. Also, depending on the type of events defined in the selected PCRs, the components of the monitored application need to be instrumented to emit the necessary runtime events into the stream observed by the Pattern Compliance Monitor. For example, in case of the Rate Limiting PCR, each HTTP request needs to be reported as an event that comprise an identifier of the client that sent the request, e.g., an IP address or access token and the response code of the request. The instrumentation can be achieved by different mechanisms depending on technology specific details of the monitored application. Thereby, Instrumentation Templates linked to the selected PCRs can be used to reduce the instrumentation effort. For example, to instrument a Java application using Aspect-oriented Programming [10], a template can be used that implements the functionality for emitting a certain event type in an aspect written in Java's aspect-oriented extension AspectJ. Aspects can be easily added to the existing code of the to be monitored application without modifying the application code itself. As another example, in case a service mesh infrastructure layer is used, a template can be used that provides configuration code for the service mesh to create a log entry for each request sent between application components. The logs can then be aggregated and sent to the Pattern Compliance Monitor.

3.2 System Architecture

The system architecture of the Pattern Compliance Monitor is depicted in Figure 3. A Web UI provides access to the functionality of the Pattern Compliance Monitor. The business logic layer comprises the five major components *CEP Engine*, *Violation Subscriber*, *Event Handler*, *PCR Manager*, and *Instrumentation Template Manager*. The CEP Engine implements a complex event processing engine that can be configured based on Stream Query Language statements to analyze a series of events. The PCR Manager is responsible for retrieving application-agnostic Pattern Compliance Rules stored in the *Pattern Compliance Rule repository*, binding them to application-specific details as described in Section 3.1, and configuring the CEP Engine using the resulting application-specific Pattern Compliance Rules. The Event Handler provides the functionality for consuming events from a given destination, e.g., a message queue, and

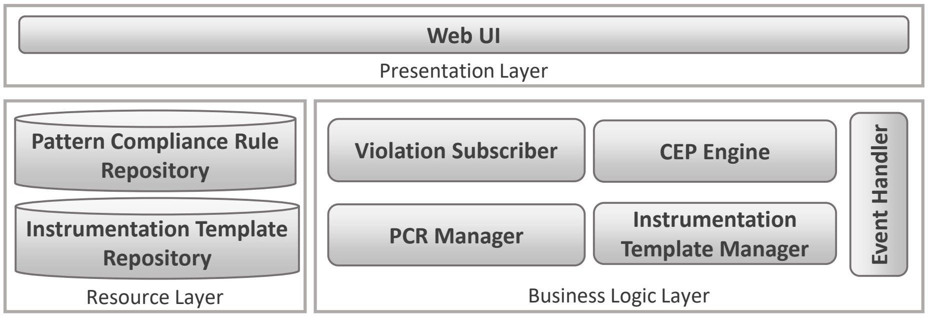

Fig. 3. The system architecture of the Pattern Compliance Monitor

adding them to the event stream analyzed by the CEP Engine. The Violation Subscriber subscribes to the Pattern Compliance Rules used as configuration for the CEP engine and receives updates about compliance violations detected by the engine. The Instrumentation Template Manager is used to manage Instrumentation Templates stored in the *Instrumentation Template Repository* and to retrieve a set of Instrumentation Templates based on the chosen Pattern Compliance Rules and technology-specific details of the application to be monitored.

4 Applying the Approach to the Motivating Scenario

In this section, the presented approach is applied to the motivating scenario described in Section 2.2. We will discuss how the behavioral aspects contained in the textual description of the patterns Circuit Breaker and Watchdog can be specified as application-agnostic PCRs using EPL statements that serve as configuration code for architecture compliance monitoring. We point out that the application-agnostic PCR for the Rate Limiting pattern is described in Section 3.1. Furthermore, we will discuss how we validated our approach of architecture compliance monitoring using the created Pattern Compliance Rules. We describe how we prototypically implemented the Pattern Compliance Monitor and instrumented a microservice-based application to be monitored to emit the necessary runtime events.

4.1 Circuit Breaker

The Circuit Breaker is a common architectural pattern used in microservice-based applications. It describes how to avoid cascading failures by wrapping functions that call remote services with a proxy that monitors failures and reacts similarly to an electrical circuit. The pattern states that when the number of consecutive failures crosses a given threshold the circuit breaker needs to trip and for the duration of a timeout period all attempts to invoke the function will fail immediately [13]. Conversely, the Circuit Breaker pattern is violated in case, even though the number of consecutive failed attempts to call a remote service is exceeded, the defined timeout is ignored and calls to the remote service are still executed. An application-agnostic Circuit Breaker PCR that describes this violation using EPL statements is shown in Listing 1.3. There, each call to a remote service is represented by an *HttpRequest* event containing the event properties *source*, which identifies the wrapped function the request originates from, and *responseCode*, which provides the returned response code of the request (line 1). In our example, we distinguish between the response code 200, which means that the request has succeeded and response code 503, which means that the request failed due to unavailability of the remote service. First, to monitor the behavior of each circuit breaker in an application separately, the HttpRequest events in the observed event stream are partitioned based on their source (line 3). Referring to this partition, a complex event called *FailureRateExcessEvent* is emitted if there is a consecutive sequence of failed HTTP requests that exceed the threshold defined by the variable *failureThreshold* (line 5-7). In other words, the event is emitted in case the defined threshold of consecutive failures for a particular remote function call is exceeded. Finally, a violation statement is defined indicating a non-compliant behavior of a circuit breaker (line 9-12). The violation is emitted if a FailureRateExcessEvent is followed by a HttpRequest event within the period defined by the variable *timeout*. This means that even though the number of consecutive failed attempts to call a remote service is exceeded, the defined timeout is ignored and calls to the remote service are still executed. Hence, the behavior described by the Circuit Breaker pattern is violated. The concrete timeout and failure threshold for the rule can be set based on application-specific decisions. For example, in the case of the architectural design decision (ADD02), described in the motivating scenario, the failure threshold is set to three and the timeout is specified as five seconds.

```
 1  create schema HttpRequest(source, responseCode)
 2
 3  create context SegmentedByCB partition by source from HttpRequest;
 4
 5  context SegmentedByCB insert into FailureRateExcessEvent
 6  select * from HttpRequest#length(failureThreshold + 1)
 7  where statusCode = "503" having count(*) > failureThreshold;
 8
 9  @Name('Timeout Violation')
10  select b.id as circuitBreakerId from pattern
11  [every (a = FailureRateExcessEvent −> b=HttpRequest(id = a.id)
12  where timer:within(timeout msec))];
```

Listing 1.3. An exemplary application-agnostic PCR for the Circuit Breaker pattern.

4.2 Watchdog

The Watchdog pattern describes a component that detects and replaces failing application component instances automatically to ensure sufficient availability of the application [7]. The pattern states that failing application component instances have to be replaced in case of failures. One possible information source for detecting failures are periodic heartbeats sent by the instances that verify proper functioning. Listing 1.4 depicts an application-agnostic Watchdog PCR that analyzes the periodic heartbeats sent by application components to monitor if failing application components are detected and replaced. Each Heartbeat sent by an application component is represented by a *Heartbeat* event, containing the event properties *id*, which identifies the running instance of an application component that sent the heartbeat, and *groupId*, which defines a logical grouping of running instances to identify replicas of a component instance (line 1). The second statement (line 3-4) defines an *InstanceCount* event to count the amount of uniquely identifiable running instances for a given replica group, i.e., the current number of replicas of a component instance. To filter out terminated instances, the statement only counts events that are emitted less than five seconds ago. The next definition statement (6-7) emits a *DecreaseCountEvent* if there is a sequence of two InstanceCount events where the first event contains a higher number as the second event. In other words, every time the amount of running instances in a replica group is decreased, a DecreaseCountEvent is emitted. Similar, an *IncreaseCountEvent* is defined which is emitted every time the amount of running instances is increased (line 9-10). Finally, a violation statement is defined (line 12-14), emitting an event that indicates a non-compliant behavior of the Watchdog component. This event is emitted if a DecreaseCountEvent is not followed by an IncreaseCountEvent within a given time threshold, i.e., failed component instances are not replaced within a certain time. The variables *monitoredGroupId* and *timeThreshold* allow customization of the statements and can be defined based on application-specific data, e.g., contained in architectural design decisions.

```
 1  create schema Heartbeat(id, groupId)
 2
 3  insert into InstanceCount select count(*) as number
 4  from Heartbeat(groupId = monitoredGroupId).std:unique(id)#time_batch(5 sec);
 5
 6  insert into DecreaseCountEvent select a.number as number
 7  from pattern [every a=InstanceCount -> b=InstanceCount(a.number > b.number)];
 8
 9  insert into IncreaseCountEvent select a.number as number
10  from pattern [every a=InstanceCount -> b=InstanceCount(a.number < b.number)];
11
12  @Name('Watchdog Violation')
13  select * from pattern [every DecreaseCountEvent ->
14  not IncreaseCountEvent and timer:interval(timeThreshold msec)];
```

Listing 1.4. An exemplary application-agnostic PCR for the Watchdog pattern.

4.3 Prototypical Implementation

For the validation of our approach, we prototypically implemented the system architecture of the Pattern Compliance Monitor described in Section 3.2 using Java and Esper[3]. We used the message broker software RabbitMQ[4] as a messaging layer. The application to be monitored can send events as messages to the broker.

The Event Handler component of the Pattern Compliance Monitor listens for new messages sent to the broker and adds them to the event stream observed by the CEP Engine. For the test setup, we have created Pattern Compliance Rules for the three patterns Rate Limiting, Watchdog, and Circuit Breaker as described in Section 3 and 4. We have evaluated the feasibility of architecture compliance monitoring using the created Pattern Compliance Rules based on both, automated unit tests simulating a synthetic data set of events and a manual test based on a prototypical implementation of a microservice application using Java. We used Java's aspect-oriented extension AspectJ for realizing unified logging of run-time events without modifying the application code itself. We created aspects for logging the run-time events described by the Pattern Compliance Rules, which were then woven into the application code. Advices in the aspects implemented the logging functionality and pointcuts associated with the advices defined the execution points at which the they should run. For example, Listing 1.5 shows an excerpt of the aspect that implements unified logging of HTTP requests as events. The aspect defines an *@AfterThrowing* advice that generates a log entry for a failed HTTP request in case a method of the application's REST client does not complete normally and an HttpStatusCodeException is thrown. Similarly, an *@AfterReturning* advice was implemented in the aspect that generates a log entry for each succeeded HTTP request.

```
@AfterThrowing(pointcut = "execution(restclient.*.*(..))", throwing = "exception")
void after(HttpStatusCodeException exception) throws Throwable {
  generateHttpRequestEvent(exception);
}
```

Listing 1.5. Excerpt of the Aspect implemented to provide unified logging of HTTP requests.

We used Logback as a logging framework and added a configuration for pushing all logs necessary for monitoring to the RabbitMQ message broker. For the deployment, we packaged each application component as a docker[5] container image. Docker Swarm was used to deploy the application as a multi-container docker application and scale services of the application during runtime. During runtime of the application we caused compliance violations by manually changing the behavioral aspects of the application that are concerned with the realization of architectural aspects described by the aforementioned patterns.

[3] https://github.com/ckrieger/ADDComplianceChecking
[5] https://www.docker.com/

5 Related Work

Different works are concerned with behavioral compliance checking or monitoring of software systems. Mulo et al. [12] propose a compliance monitoring approach for verifying that business processes adhere to specified compliance controls. They provide a DSL that can be used to define compliance monitoring directives based on business activities and translate them into an event-based sequence that serve as compliance monitoring code. Similar to our work, complex event processing is used to implement the approach. However, their work focus on compliance concerns in the context of laws and regulations, whereas our work is concerned with checking behavioral compliance with architectural patterns. Ackermann et al. [1] compare UML sequence diagrams, which describe the intended interaction of components in a software system, against the actual behavior implemented in the system to construct a behavioral reflexion model that shows potential drift between the desired behavior of a system and the actual implementation. In contrast to ours, their work does not focus on constraints described by architectural patterns. Wendehals et al. [19] present an approach to recognize behavioral aspects of object-oriented design patterns in legacy systems by instrumenting relevant method calls and monitoring them at runtime. In contrast to our work, they use finite automata to describe behavioral aspects of the patterns. Moreover, their work focus on the detection of design patterns in object-oriented software, whereas, our work focus on architectural patterns that are relevant in the domain of distributed cloud applications. Breitenbücher [3] proposes the formalization of management patterns, e.g., patterns for the management of cloud applications, to allow their automated execution for individual applications. In contrast to ours, this work is not concerned with automated compliance monitoring but with the automated execution of the management steps described in the patterns. Saatkamp et al.[17] propose to formalize the knowledge contained in architecture and design patterns to automatically detect problems in restructured topology-based deployment models. The formalization and automated detection are based on the logic programming language Prolog. This approach is concerned with structural problem detection during design time of deployment models. In contrast, our work monitors behavioral compliance during the runtime of an application. Fahland et al. [5] provide a formalization of Enterprise Integration Patterns based on Coloured Petri Nets. Similar work is presented by Ritter et al. [15]. They propose a new formalism called timed db-nets to formally describe Enterprise Integration Patterns. Both works are exclusively concerned with the formalization of Enterprise Integration Patterns, whereas our work describes a general approach to define behavioral directives described in architectural patterns. Related in the broader sense is the work by Falkenthal et al. [6]. They introduce the concept of Solution Implementations as concrete implementations of patterns. Selection Criteria added to the relations between Solution Implementations and patterns allow to determine the most appropriate implementation for a specific use case. The concept of Solution Implementations can be used to provide a set of reusable compliant pattern implementations.

6 Conclusion and Future Work

Monitoring the system for architectural compliance helps to quickly detect inconsistencies between the intended behavior and its actual implementation. In this paper, we proposed to apply behavioral directives described by architectural patterns as input for architecture compliance monitoring of an application. For this, we presented (1) how stream query languages can be used to specify the intended behavior described in architectural patterns using runtime events, (2) how the resulting rules can be transformed into application-specific, machine-processable instructions, (3) and how a system can be automatically monitored for behavioral compliance violations using such rules. We applied the presented rule configuration approach to the Rate Limiting, Circuit Breaker, and Watchdog pattern. Further, for validating the feasibility of our approach, we prototypically implemented a Pattern Compliance Monitor based on Esper's complex event processing engine. The presented approach is not limited to the discussed patterns and can be extended to monitor compliance with other architectural patterns. However, one possible limitation to this is that some architectural patterns might comprise insufficient behavior or the described behavioral directives cannot be sufficiently expressed using stream query languages. In future work, we plan to investigate different pattern languages to identify suitable architectural patterns for extending the presented approach. Another limitation is that the approach presumes knowledge about stream query languages, e.g., the Esper EPL. Translating the intended behavior described in patterns into EPL statements can become a complex and non-trivial task. It is left for future work to ease this process, e.g., by developing a domain-specific language and tool support for the creation of Pattern Compliance Rules.

ACKNOWLEDGMENTS

This work was partially funded by the DFG project ADDCompliance (636503), the European Union's Horizon 2020 research and innovation project RADON (825040), FWF (Austrian Science Fund) project ADDCompliance: I 2885-N33, and FFG (Austrian Research Promotion Agency) project DECO, no. 846707.

References

1. Ackermann, C., et al.: Towards behavioral reflexion models. In: Software Reliability Engineering, 2009. ISSRE'09. 20th International Symposium on. pp. 175–184. IEEE (Nov 2009)
2. Alexander, C., et al.: A Pattern Language: Towns, Buildings, Construction. Oxford University Press (Aug 1977)
3. Breitenbücher, U.: Eine musterbasierte Methode zur Automatisierung des Anwendungsmanagements. Dissertation, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik (2016)
4. EsperTech: Esper (2019), http://www.espertech.com/esper/
5. Fahland, D., Gierds, C.: Analyzing and completing middleware designs for enterprise integration using coloured petri nets. In: International Conference on Advanced Information Systems Engineering. pp. 400–416. Springer (Jun 2013)

6. Falkenthal, M., et al.: From Pattern Languages to Solution Implementations. In: Proceedings of the 6th International Conferences on Pervasive Patterns and Applications (PATTERNS 2014). pp. 12–21. Xpert Publishing Services (May 2014)
7. Fehling, C., et al.: Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications. Springer (Jan 2014)
8. Hackernoon: Designing a microserices architecture for failure (2017), hackernoon.com/designing-a-microservices-architecture-for-failure-a57f34ded646
9. Harrison, N., et al.: Using Patterns to Capture Architectural Decisions. Software **24**(4), 38–45 (Jul 2007)
10. Kiczales, G., et al.: Aspect-oriented programming. In: European conference on object-oriented programming. pp. 220–242. Springer (1997)
11. Luckham, D.: The power of events: An introduction to complex event processing in distributed enterprise systems. In: International Workshop on Rules and Rule Markup Languages for the Semantic Web. pp. 3–3. Springer (2008)
12. Mulo, E., et al.: Domain-specific language for event-based compliance monitoring in process-driven soas. Service Oriented Computing and Applications **7**(1), 59–73 (Mar 2013)
13. Nygard, M.T.: Release it!: design and deploy production-ready software. Pragmatic Bookshelf (Mar 2007)
14. Oppenheimer, D.: The importance of understanding distributed system configuration. In: Proceedings of the 2003 Conference on Human Factors in Computer Systems workshop. Citeseer (2003)
15. Ritter, D., et al.: Formalizing application integration patterns. In: 2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC). pp. 11–20. IEEE (2018)
16. Rosik, J., et al.: Assessing architectural drift in commercial software development: a case study. Software: Practice and Experience **41**(1), 63–86 (Aug 2011)
17. Saatkamp, K., et al.: An approach to automatically detect problems in restructured deployment models based on formalizing architecture and design patterns. SICS Software-Intensive Cyber-Physical Systems pp. 1–13 (Feb 2019)
18. Stocker, M., et al.: Interface quality patterns – communicating and improving the quality of microservices apis. In: 23rd European Conference on Pattern Languages of Programs 2018 (Jul 2018)
19. Wendehals, L., Orso, A.: Recognizing behavioral patterns at runtime using finite automata. In: Proceedings of the 2006 international workshop on Dynamic systems analysis. pp. 33–40. ACM (May 2006)
20. Zdun, U., et al.: Ensuring and assessing architecture conformance to microservice decomposition patterns. In: International Conference on Service-Oriented Computing. pp. 411–429. Springer (2017)
21. Zimmermann, O., et al.: The role of architectural decisions in model-driven service-oriented architecture construction. In: Proceedings of the OOPSLA 2006 Workshop on Best Practices and Methodologies in Service-Oriented Architectures, Unipub (2006)
22. Zimmermann, O., et al.: Reusable architectural decision models for enterprise application development. In: International Conference on the Quality of Software Architectures. pp. 15–32. Springer (Jul 2007)
23. Zimmermann, O., et al.: Combining pattern languages and reusable architectural decision models into a comprehensive and comprehensible design method. In: Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008). pp. 157–166. IEEE (Feb 2008)