Static and Dynamic Architecture Conformance Checking: A Systematic, Case Study-Based Analysis on Tradeoffs and Synergies

Jan Thomas
RWTH Aachen University
jan.thomas@rwth-aachen.de

Ana Nicolaescu
RWTH Aachen University
ana.nicolaescu@swc.rwth-aachen.de

Horst Lichter
RWTH Aachen University
horst.lichter@swc.rwth-aachen.de

***Abstract*—In order to uncover architectural drift, a plethora of architecture conformance checking tools has been proposed that mainly leverage two approaches: they extract architectural knowledge based on either source code artifacts (static approach) or run-time behavior (dynamic approach). Although both approaches have been evaluated separately, no up-to-date analysis of their relative strengths and weaknesses, nor real-world comparative case studies of the two were published. In this paper we address this issue by presenting the results of a direct comparison of both approaches. We first identify and compare their strengths and weaknesses on a theoretical level. We then evaluate these results against our experiences gained in a large-scale industrial case study. As a result, we argue that the approaches cannot substitute each other as they differ in many key aspects. Hence, we crystallize guidelines regarding how to combine these such that their strengths are emphasized while weaknesses mitigated.**

I. Introduction

It is well studied and understood [1] [2] that the quality of a software system's architecture directly influences its non-functional properties (NFP) like understandability, maintainability, or security. As a consequence, developing successful large-scale software solutions requires a well-documented architecture. If the documented architecture accurately reflects the built system, it serves as the key artifact for architectural communication among different stakeholders and sound architectural decision making [3]. However, as software systems evolve, it is often the case that a gap between the actually implemented architecture (as-implemented) and its documentation (as-intended) emerges. This gap is referred to as architectural drift [4]. If this drift is not addressed properly, it could cause the violation of architectural design decisions (architectural erosion) [5], which can cause serious problems related to the NFPs listed above.

In order to uncover architectural drift, several architecture conformance checking techniques were proposed [6] (e.g. reflexion modeling [7]) and implemented in commercial [8] [9] and non-commercial [10] tools. These tools collect architectural evidence from a system's source code and compare it to an as-intended architecture (static approach). However, current research shows that automated architecture conformance checking techniques and tools are still not well adopted by the industry [11] [12].

In addition, with the advent of new architecture styles (e.g. microservices), the complexity of software systems shifts from their static structure to their run-time behavior. Consequently, static analysis tools might not cover all architecturally relevant aspects. To address this issue, tools were proposed [13] [14] [15] that collect architectural evidence by means of run-time data and therefore cover behavioral aspects as well (dynamic approach).

Although both approaches have been discussed separately, no work addressed a direct comparison thereof. Guided by this observation, we derived two research questions:

- **RQ1** - Considering their general capabilities, which are the main strengths and weaknesses of static and dynamic architecture conformance checking approaches?
- **RQ2** - How can both approaches be combined in order to obtain synergies?

Addressing these questions on a theoretical foundation can be done by consulting existing literature and related work. However, answering such questions outside the scope of a real-world context is not recommended, as results may lack applicability and industrial relevance. Hence, we also conducted a case study as a six months long project that emerged within the cooperation with one of our industry partners. During this case study, we explored the applicability of both architecture conformance checking approaches in a large-scale industrial context. As a result, the key contributions of this paper are bivalent. (1) We contribute a theoretical comparison of static and dynamic approaches in general and (2) undergo a comparison based on the experiences gathered from our industrial case study. Both are important contributions towards answering the research questions as defined before.

The remainder of this paper is structured as follows: Section II investigates the theoretical background of static and dynamic approaches. Subsequently, we introduce the context and design of our case study in Section III. Next, we present the challenges that we encountered while performing the case study and continue with the evaluation of the obtained results. A thorough comparison and discussion of both approaches based on the previous results is subject to Section IV. Section V presents related work. Last, Section VI concludes the paper with a summary and a discussion of future work.

II. Background of Static and Dynamic Approaches

In comparison to other techniques for checking architecture conformance, reflexion modeling has a large industrial acceptance and a great maturity of tool support [16]. Originally, reflexion modeling was introduced [7] as a technique to uncover architectural drift based on three prerequisites: an as-intended architecture model, low-level architectural evidence and a mapping between both. Based on these inputs, entities and relations of the as-intended architecture model are identified as architectural convergences, divergences or absences. The capabilities of reflexion modeling strongly depend on the type of architectural evidence (entities and relations) that can be extracted by either analyzing source code (static approach) or by analyzing run-time data from an instrumented system (dynamic approach).

A. Architectural Entities and Relations

We illustrate which entities and relations can be obtained for both approaches on the example of Java. Analyzing Java source code, one can obtain three kinds of entities: files, packages and types (classes, enums or interfaces). Each source file defines a package that it *belongs* to and defines a set of *imported* types. Classes can *extend* other classes and *implement* interfaces. Interfaces are able to *extend* other interfaces. Within classes, a different set of relations can be found. Classes can be *instantiated*, variables of classes or instances can be *accessed* and methods of classes or instances can be *invoked*. While the *belongs to package* relation defines the structure of entities, all remaining ones (import, extend, implement, instantiate, variable access, method invocation) express different kinds of usage. As it is possible to define these relations during run-time[1], it is possible that the resulting relations are not created ahead of run-time. Hence, usage that is triggered in this dynamic manner is not within the scope of the static approach.

In contrast, applying the dynamic approach one needs to extract architectural evidence through instrumentation of a system using special monitoring tools like Dynatrace [17] and triggering its behavior (e.g. running test-cases), which we refer to as *episodes*. The monitoring tools intercept and capture method invocations at run-time and yield a set of so-called execution traces. A trace can be defined as a tree of caller-callee-nodes. Each node includes the name of the invoked method and the fully qualified name of the type it belongs to. Furthermore, nodes can be augmented with additional run-time information (e.g. object ids or method arguments). In addition, the execution frequency of a method can be derived from all observed invocations. Dependent on the utilized monitoring tool, further relations are in the scope of the dynamic approach. This includes method invocations which are triggered dynamically (e.g. by reflection) as well as variable access. Furthermore, relations resulting from inter-process communication (e.g. pipes, database access, or web services) can also be extracted. Table I summarizes the previous analyzed architectural relations. Based on the architectural evidence which can be obtained and how it is extracted, we can derive the strengths and weaknesses of either approaches.

[1]using techniques such as reflection and dependency injection

TABLE I
Relations obtained from architectural evidence

Relation Type	Static Appr.	Dynamic Appr.
Import	x	
Extends	x	
Implements	x	
Variable Access	x	x
Instantiation	x	x
Method Invocation	x	x
Dynamic Method Invocation		x
Inter-Process Communication		x
Execution Frequency		x
Execution Time		x

Using reflexion modeling techniques, the as-intended architecture can be expressed as rules of the type "*is allowed to use*" (e.g. component A is allowed to use component B). The type of usage can be refined based on the extracted architectural relation types. Compared to some basic usage rules, which can be formulated in both approaches, more complex ones can be defined based on the relations obtained by the dynamic approach. First, it can distinguish multiple method invocation types (e.g. direct invocation, database access, remote procedure calls, web services, message bus). Second, communication parameters captured at run-time (e.g. method invocation arguments or web service endpoint) can also be considered when modeling usage-based rules. For instance, architects could define rules that restrict access to a component through a specific REST API. Similarly, given a component, the architects can choose to restrict its database access to a specific table. Third, besides *is allowed to use* rules, the dynamic approach also facilitates the definition of rules based on the temporal order of captured method calls or their frequency. For instance, one could define rules stating that a method call A must happen before B or that A must not be called more than once. Generally speaking, the dynamic approach facilitates the analysis of richer architectural properties.

B. Distinguishing Capabilities

A key benefit of the dynamic approach is the ability to analyze **inter-process communication**. This becomes essential when analyzing systems following new architecture styles like microservices. The complexity of these systems shifts from their static structure to their interaction and behavior at run-time, as they are distributed across multiple processes. In these cases, only the dynamic approach can provide insights about the entire system, as the concrete architecture is only assembled at run-time.

Even if architectural conformance analysis is restricted to homogeneous single process systems, not all architecturally relevant information can be obtained from the source code [3]. For instance, relations caused by **late binding** (e.g. polymorphism) can only be extracted from running systems. However, late binding effects might not always be desired.

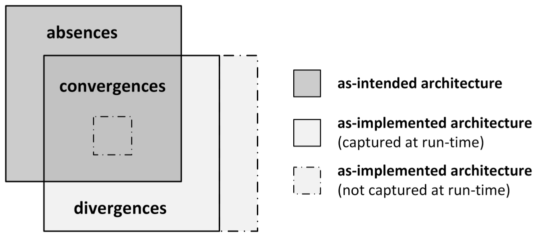
absences
convergences
divergences
as-intended architecture
as-implemented architecture (captured at run-time)
as-implemented architecture (not captured at run-time)

Fig. 1. Reflexion models - Application on the dynamic approach

For instance, interfaces are often used as a measure to prevent direct coupling between classes. While the static approach is aware of this decoupling, the dynamic approach is not because it observes the dynamic, late bound type at run-time. This anomaly needs to be considered by architects when modeling rules for either the static or dynamic approach.

A major difference between the static and dynamic approach is the analysis **completeness** [18]. While architectural evidence extracted from source code holds for all program executions, architectural evidence obtained from run-time data only holds for the observed behavior. As a consequence, the dynamic approach cannot guarantee that a system satisfies a particular architectural property, but can only detect violations of those. In contrast, the static approach gives this guarantee, as its properties hold for all program executions. In case that both approaches disagree regarding a certain property, either the dynamic approach did not cover all relevant parts of the system or the static approach analyzed code that is either unreachable or not used anymore [18]. As the dynamic approach considers actual program executions, it does not suffer from analyzing unreachable or unused code. However, due to the same reason it is hard to achieve completeness.

Applying these thoughts regarding completeness to reflexion modeling techniques, adds a fuzziness to the results of the dynamic approach (cf. Figure 1). Parts of the as-implemented architecture that were not captured due to incomplete instrumentation are represented as dashed rectangles. Taking the possibility of uncovered parts into account leads to three considerations. First, if not all parts of the as-implemented architecture are covered, it is not certain that the set of divergences reflect all actual divergences in the software system. There might be architectural violations in the system which are not detected, thus this set only suites as a lower bound. Second, absences may be detected that actually are convergences due to uncovered parts in the as-implemented architecture. Therefore, the set of detected absences can only be interpreted as an upper bound, whereas the set of detected convergences represents a lower bound. Third, the probability that these match the actual sets in the software system is influenced by the degree of uncovered parts in the as-implemented architecture. If a large uncovered fraction exists, this probability is low. In contrast, lowering this fraction increases the robustness and reliability of the results. As a consequence, architects need to make a reasonable episode selection such that all parts of the system that are relevant for the analysis are covered.

TABLE II
SYSTEM UNDER ANALYSIS (SUA) - SIZE-ORIENTED METRICS

Metric	Value
Lines of code	$\sim$ 125000
Number of Apache Maven projects	51
Number of OSGi bundles	30
Number of packages	138
Number of source files	879

In order to make a reasonable episode selection, one could follow guidelines to ensure that these cover the system's most relevant use-cases. Besides using such guidelines, the quality of selected episodes should also be measured objectively. To this end, architects can adopt structural coverage metrics (e.g. statement coverage) in the context of architectural conformance checking to assess the quality of selected episodes (see e.g. [19] or [20]). Measuring coverage metrics bears the advantage of gaining confidence regarding the analysis significance. In addition, these metrics are easy to measure due to mature tool support for major programming languages.

To sum up, in this section we defined the analysis scope of both approaches based on entities and relations obtained from architectural evidence. In addition, we distinguished the approaches by capabilities of analyzing interprocess communication, late binding effects and characteristics regarding completeness.

III. CASE STUDY

This section describes the conducted case study. We first describe the case study context and design, before we comment on encountered challenges and present the obtained results.

A. System Under Analysis (SUA)

Subject to this case study is a system for task automation and data distribution, which is developed since 2008. As part of the software evolution process and driven by customer needs, features were constantly added and improved. As a consequence, its architecture was refactored several times. Today, the Java OSGi based system comprises a total of 125000 lines of code. Table II summarizes additional size-oriented metrics. As the SUA represents a business critical component, it is crucial to ensure that all non-functional properties considered by the architects are actually respected in the implementation. Hence, there is a strong need for automated architectural conformance checks.

The system is divided into five processes, each implementing an individual functional slice. Each process runs an OSGi container which hosts several OSGi bundles. These in turn can provide or require OSGi services and depend on other *.jar* artifacts. Persistence is realized through a RDBMS and local XML files. Inter-process communication (IPC) among the five processes is implemented by polling for state changes in the persistence layer as well as by publishing and subscribing to a message bus. As data distribution is a central feature of the SUA, it implements several communication protocols

TABLE III
SUA - ARCHITECTURE DIAGRAMS

Diagram	Concepts	Dependencies
System	Processes, Message Bus	External Protocols, File System
Process	OSGi Bundles	External Protocols, File System
OSGi Bundle		External Protocols, File System, .jar artifacts, OSGi Services

(e.g. FTP, HTTP, etc.). Its architecture is documented in a set of UML component diagrams which depict several levels of abstractions: system, process and OSGi bundle. Each diagram models how the inner concepts are connected and defines dependencies to the context in which it is embedded. Table III depicts which concepts and dependencies are represented in which diagram type.

Based on these diagrams and expert interviews, we derived properties that had to be analyzed in the case study: (1) Do the OSGi bundles use just the documented *.jar* dependencies? (2) Are OSGi services wired at run-time as documented? (3) Is the database used only through the access layer component? (4) Are only the allowed components coupled at run-time through the message bus? (5) Are only white-listed directories and files accessed at run-time? (6) Are external systems always used through their dedicated protocol facades? While properties 1, 3 and 6 are related to the static structure of the SUA, the analysis of property 2, 4 and 5 requires insights into the system's run-time. For this reason, we decided to combine the tools Sonargraph-Architect (static approach) and ARAMIS (dynamic approach) for this case study.

B. Employed Static Tool - Sonargraph-Architect

Sonargraph-Architect [8] is a well established tool for static code analysis. It allows to monitor complex software systems regarding their technical quality and to enforce architectural conformance rules. In addition, it calculates a variety of metrics. While Sonargraph supports many features, we just used the architecture conformance related features for this case study. Sonargraph-Architect provides a domain-specific language (DSL) to define a system's architecture. According to the principles of reflexion modeling, the DSL supports the definition of components which are mapped to source code and connected by architectural rules (e.g. allowed to instantiate, allowed to call, etc.).

C. Employed Dynamic Tool - ARAMIS

ARAMIS (Architecture Analysis and Monitoring Infrastructure) is a tool-supported framework for run-time monitoring, communication integrity validation, evaluation and visualization of the behavior view of software architectures [15]. In order to automatically check for architectural conformance using ARAMIS, three steps need to be followed. First, an as-intended architecture description needs to be defined for the SUA by using ARAMIS' dedicated architecture description language (ADL). This description includes a hierarchy of so-called architecture units, their mapping to the source code elements and their allowed interaction rules. Second, the SUA needs to be instrumented while performing certain episodes in order to capture low-level architectural evidence in form of run-time traces. As ARAMIS builds on top of well established monitoring infrastructures, we used the Dynatrace [17] monitoring infrastructure, as it is able to monitor complex distributed and heterogeneous systems. Third, the captured traces need to be analyzed by the ARAMIS processing chain regarding architectural violations.

D. Case Study Design and Execution

During an initialization phase, we organized several meetings to discuss the general setting and high-level requirements for automated architecture conformance checks. Afterward, the source code (*as-implemented architecture*) and the diagrams (*reference as-intended architecture*) were manually inspected. Based on this inspection and expert interviews, architectural questions of interest tailored to the SUA were derived. As the utilized tools can not directly perform conformance checks based on the reference model, *tool-specific as-intended architecture* models needed to be created. We decided to exercise an automated model-to-model transformation approach which was studied in our previous work [21]. This approach enabled us to obtain tool-specific models of comparable quality, which facilitates a fair comparison of the static and dynamic approach. Another prerequisite for applying the tools is the availability of *architectural evidence*. While the source code could be accessed easily by Sonargraph-Architect, we had to instrument the SUA and capture traces at run-time for ARAMIS using Dynatrace. To this end, we used an already existing UI test suite and measured its statement coverage to assess its adequacy for supporting a behavior-based conformance check. Using the automatically transformed models and the architectural evidence, we then applied the tools Sonargraph-Architect and ARAMIS on the SUA in order to uncover its architectural drift. Subsequently, we manually verified our results by classifying the detected architectural violations into defects in the as-intended architecture, defects in the as-implemented architecture and false positive violations. This activity was conducted in cooperation with architects and developers.

E. Challenges

This section illustrates challenges we encountered when conducting the case study and explains countermeasures we took.

As stated above, we followed a model-to-model transformation approach to obtain tool-specific as-intended architecture models from a set of reference as-intended architecture diagrams. When implementing this transformation, we encountered two major challenges. First, we needed to automatically map components in the reference model to implemented classes or packages in the source code. Second, the diagrams denoted connections between components in terms of Java interfaces. While the static approach is aware of interface decoupling, the dynamic approach is not due to late binding. As traces captured at run-time do not state usage of an interface, but usage of a concrete implementation, rules in the dynamic approach need to be aware of the concrete implementations. However, as these were not denoted in the reference diagrams, it was not possible to automatically generate these rules without any additional information. The presented issues boil down to two questions: (1) Which packages and classes are defined within a component given its name? (2) Which classes implement a particular interface given its name? In order to automatically answer these questions, we decided to embed a static pre-analysis of the component's source code into the transformation process. This analysis creates two mappings by iterating over all components and their classes: (1) component name $\Rightarrow$ included packages and classes names, (2) interface name $\Rightarrow$ implementations' class names. By using these mappings, the previous questions could automatically be answered, which enabled us to apply the planned model-to-model transformation approach.

As usual with dynamic approaches, scalability issues needed to be carefully addressed. The monitoring of the approximately 140 test cases lasted for 33 hours and produced 16 GB of trace information comprising over 35 million caller-callee-pairs. To reduce the amount of data, we applied our knowledge regarding the monitored tests and the system's general architecture: redundant traces result through (1) test fixture setup and tear down and (2) periodic polling employed by the processes. We discarded these duplicate traces by utilizing heuristic-based data deduplication techniques. This lead to a comprehensible reduction of over 60% of the data to be analyzed and its processing time.

F. Results

When exercising both approaches, we obtained 20 types of architectural violations using ARAMIS and 15 violations using Sonargraph-Architect. Only 3 of these violation types were detected by both tools. With the help of architects, we further classified these into defects in the as-intended architecture, defects in the as-implemented architecture and false positive violations. About two fifth were identified as false positives due to false mapped classes or due to false modeled rules. The majority of the remaining violations was traced back to imprecise information in the reference as-intended architecture model. Sonargraph-Architect uncovered undocumented external dependencies, which were not in the scope of ARAMIS as we reduced its instrumentation scope. In addition, Sonargraph-Architect detected 4 architectural violations, which were not detected by ARAMIS due to insufficient coverage produced by the monitored test suite. Both observations confirm our previous thoughts on completeness of the dynamic approach. In contrast, ARAMIS was able to uncover file system related violations which were not in the scope of Sonargraph-Architect. In addition, we identified an unused database connection and two performance critical violations by analyzing the execution frequency captured at run-time. These valuable run-time insights highlight the unique capabilities of the dynamic approach.

Although we monitored about 140 automated UI test cases, only 32% of all statements in the SUA were covered at run-time. While this low coverage does not give any confidence that we captured a relevant part of the SUA, the detailed coverage report gives valuable hints how the utilized test suite could be improved (e.g., only 26% of the system's public API were covered).

Once set up, an important performance indicator of architecture conformance checks is the cycle time for the whole analysis process. We experienced cycle times of less than a minute for the static approach. In contrast, the dynamic approach was a long running process which took 13 hours (deduplicated traces) up to 33 hours (full traces).

IV. Comparison and Discussion

In the previous sections we mainly elaborated on the basic characteristics of both approaches (RQ1). In contrast, this section first consolidates the previous results for a comparison of both approaches. Subsequently, we discuss how both approaches can be combined in order to obtain synergies (RQ2).

A. Comparison

Knodel and Popescu derived different comparison dimensions as part of their work [6] in order to compare three different static architecture conformance checking approaches. Being generic dimensions, we reuse a subset of these and adapt them for our work. In particular, these dimensions cover required inputs, involved stakeholders, manual tasks, the analysis scope, the analysis completeness, evaluation performance, scalability factors and maintenance aspects. The following discusses our comparison results which are summarized in Table IV.

Before choosing a conformance checking approach, companies need to be aware of prerequisites in terms of required **inputs** . Both approaches require a tool-specific as-intended architecture model which can either be modeled manually or be transformed automatically from a reference model. In our case study, we augmented the reference model by static information gathered from source code. Furthermore, both conformance checking approaches depend on architectural evidence for their analysis. The static approach can infer this evidence directly from source code. In contrast, the dynamic approach needs to capture run-time traces from an instrumented system for this task. In addition, this process requires a solid instrumentation configuration and an episode selection that captures all architecturally relevant parts of the SUA. These inputs need to be available upfront the analysis and maintained throughout the systems lifetime by different stakeholders.

For both approaches, the creation of tool-specific architecture models involves manual work either by manually creating the model or by implementing a tailored model-to-model transformation. However, if a similar system was analyzed

TABLE IV
COMPARISON OF THE DYNAMIC AND STATIC APPROACH

Dimension \ Approach	Dynamic	Static
Inputs	• as-intended architecture model, source code • episode selection, instrumentation configuration	• as-intended architecture model, source code
Manual Tasks	• create initial as-intended architecture model • review results • orchestration of process steps	• create initial as-intended architecture model • review results
Analysis Scope	• heterogeneous systems • inter-process communication, late binding	• homogeneous systems • static structure
Completeness	• not complete • approximately characterized by coverage metrics	• complete
Evaluation Performance	• long running process	• instant feedback
Scalability Factors	• instrumentation overhead • resource usage increases with captured traces	• none experienced
Maintenance Subjects	• as-intended architecture models • instrumented system, episode selection	• as-intended architecture models

before, the transformation might be reused or adapted. Beyond that, the static approach does not need much manual effort to carry out the analysis process due to mature tool support. In contrast, behavior-based non-commercial tools need much more manual effort as they are not as mature. In the case of ARAMIS, trace capturing, importing and processing need to be initiated and monitored manually by architects. Another important manual task is the discussion of detected violations by architects and developers in order to identify their cause and possible solutions. As architectural violations can be caused individually by defects in the as-intended architecture, by defects in the as-implemented architecture or by defects in the analysis process (false positives), this task can not be handled by tools.

In order to choose the right approach, it is important to consider the approachs **analysis scope**. As identified before, static tools collect architectural evidence directly from source code. Hence, the scope of a particular analysis is constrained to a single programming language and in turn to homogeneous systems only. In contrast, the analysis scope of the dynamic approach mainly depends on the utilized monitoring infrastructure. Advanced ones (e.g. Dynatrace) are able to collect traces across system boundaries. Therefore, these tools facilitate the analysis of heterogeneous systems-of-systems. Architectural relations that can be analyzed by a particular approaches were illustrated in Table I. Summarizing this table, the static approach has to be used if the analysis should include rules based on import, extends or implements relations. On the contrary, the dynamic approach has to be applied if dynamic method invocation, inter-process communication, execution frequency or timing related aspects are relevant. Rules based on method invocation can be modeled in either approaches. However, if the dynamic type of an invoked method (late binding) is relevant for the analysis, the dynamic one needs to be used as the static one is not aware of late binding concepts.

A pivotal difference between the static and dynamic approach is the **analysis completeness**. The static approach is complete within its scope as it can access all architectural relations easily from source code. In contrast, the dynamic approach is not complete because in practice it is not possible to capture behavior of an application in its entirety. As a consequence, the dynamic approach cannot guarantee that a system satisfies a particular property for all program executions. As stated before, the degree of completeness depends on the quality of selected episodes. However, increasing the quality of these, such that they cover more parts of the SUA, has two major influences. First, it requires significantly more effort to write and maintain test suites for those episodes. Second, as more episodes are monitored, more traces are captured. As a consequence, the overall analysis has an increase in trace capturing time, processing time and storage requirements.

Both approaches differ significantly in their cycle time of the overall analysis process. In the context of our case study, we experienced short **cycle times** of less than a minute for the static approach and long cycle times of 13 to 33 hours for the dynamic one. Therefore, the static one offers the opportunity to provide instant feedback, whereas the dynamic one is a long running process which needs to be planned and scheduled in advance.

Finally, one needs to take **maintenance aspects** into account, before applying one or the other approach. For both approaches it is crucial to maintain the reference and tool-specific as-intended architecture for the SUA. Two additional artifacts need to be maintained for the dynamic approach. First, an instrumented version of the SUA needs to be maintained in order to capture traces. As the source code changes, its instrumentation configuration must be updated accordingly. Second, the episode selection must be maintained accordingly to changes in the source code or the architecturally relevant use cases of the system.

B. Discussion

The previous comparison points out that both approaches have their own strengths, weaknesses and unique features. Hence, one approach can not substitute the other. However, the dynamic one requires significantly more effort and resources compared to the static one in almost all dimensions. As a result, the static approach should be preferred according to the "*least effort extraction*" strategy [3] for analyzing architectural properties which are within the scope of both approaches. Beyond that, we propose to first establish a solid static architecture conformance analysis before applying the dynamic approach. According to the pareto principle, a static analysis can already identify a large fraction of architectural violations using comparatively little effort. In addition, it bears the advantage of being complete within its scope. Due to the ability of providing instant feedback, the static approach could potentially be integrated into IDEs, which has two advantages. First, it increases the architectural awareness and hence prevents the emergence of severe architectural violations at development time. Second, if it is applied frequently, it facilitates a tight feedback loop between developers and architects by continuously validating architectural decisions. In summary, the static approach should be used early and often in order to establish the foundation of tool-based architecture conformance checking.

However, using solely static tools is not sufficient for a comprehensive architecture analysis. Based on our analysis and the performed case study, we see the following two scenarios where the dynamic approach can enhance the results of a static analysis

First, if analyzing heterogeneous systems-of-systems, a dynamic analysis focusing on inter-process communication can augment the static analysis, which focuses on inner-process properties only; in particular, a static analysis needs to be augmented by a dynamic one, if architectural relevant properties, like security or performance are influenced by concepts such as late binding, etc..

Second, as applying just the static approach does not give insights how often particular violations occur at run-time, the dynamic approach can be used to measure the frequency of architectural violations in order to assess their severity.

To sum up, we propose the **combination of static an dynamic approaches** in order to achieve a broader analysis scope. The static approach should be preferred for all architectural properties where it is applicable as it is complete and requires less effort. As it provides instant feedback, it should be used frequently by developers in order to increase architectural awareness and prevent architectural drift. To complement missing architectural properties of the static analysis, a dynamic one should be conducted on demand.

V. Related Work

Architecture conformance checking has been in the focus of research for a long time. Ducasse and Pollet [22] introduced a taxonomy of the field and presented a comprehensive overview of existing tools and techniques. Static approaches were discussed in [6] [23] [10]. Approaches focusing on the behavior of software systems were studied in [13] [14] [15]. But, to the best of our knowledge, there is no work that performed a direct comparison of both approaches, based on a theoretical level and case study results.

Knodel and Popescu [6] conducted a comparison of three low-level static architecture conformance checking techniques in the context of a tool for static architecture evaluation called SAVE. Guided by a goal-question-metric approach, they derived 13 dimensions in order to compare the three techniques. We used a subset of these dimensions and adopted them for our work in section IV. Like in our work, the authors identified strength and weaknesses for each technique. They proposed that architects should individually choose the right technique for their needs based on the comparison dimensions and results.

In [24], Knodel et al. reported a long term experience on transferring static architecture conformance checking to their industry partner. They identified the need for automating the process to a large extent due to time constraints of their industry partner. In [25] Rosik et al. present their results obtained while applying static architecture reconstruction over a time-span of two years for one of their industrial partners. Among others, they recommend to apply an adaptation of the reflexion modeling technique: conformance checking should be undergone periodically during the actual development and not only on the completed system. Similarly, Ganesan, Keuler, and Nishimura [20] share their experience applying the dynamic approach in an industrial context. They experienced that conformance checking is best applied iteratively and in close cooperation with architects. Furthermore, they propose to measure the code coverage metrics of monitored use cases to ensure that all relevant architectural components were covered at run-time. This approach was also proposed by [19] and [26].

Similar to these case studies, we also provided architecture conformance checks as a service. In addition, we automated the creation of tool-specific architecture models and captured coverage metrics to assess the quality of our dynamic analysis. In contrast to these studies, we applied the static and dynamic approach in parallel. This facilitated a comprehensive comparison of both approaches and a broader architectural analysis scope.

VI. Conclusion and Future Work

In this paper, we studied the characteristics, strengths and weaknesses of static and dynamic architecture conformance checking approaches and examined how synergies can be obtained by combining both. In the following, we draw a conclusion regarding our initial research questions.

RQ1 - Considering their general capabilities, which are the main strengths and weaknesses of static and dynamic architecture conformance checking approaches? Static and dynamic approaches are contrary to each other as most strengths of one approach are the weaknesses of the other. We studied this on a theoretical foundation (cf. section II) and supported our results with an industrial case study (cf. section III). On the one hand, the static approach enables an architectural analysis which is complete within its scope spending comparable little effort. In addition, it provides instant feedback and therefore can be used frequently. On the other hand, its scope is limited to the static structure of software systems, which excludes the analysis of inter-process communication, dynamic method invocation and effects of late binding. However, these concepts are within the scope of the dynamic approach. But in contrast to the static one, it is not complete and requires significantly more effort and processing time. In turn, it is aware of the frequency violations occurred at run-time, which can be used to assess their severity.

RQ2 - How can both approaches be combined in order to obtain synergies? As both approaches are contrary to each other, one can not substitute the other. In order to achieve a broader analysis scope, we proposed a way of combining both such that their strengths are emphasized and weaknesses are mitigated. To utilize the completeness, low effort and instant feedback of the static approach, it should be preferred over the dynamic one if it is applicable and should be used frequently. If the static analysis does not cover all relevant properties and the project resources permit, a dynamic analysis can be used to cover the missing properties. As it is not complete, structural coverage metrics should be measured in conjunction to the analysis in order to expose uncovered parts of the analyzed system. Due to its high effort and processing time, dynamic architecture conformance checks should only be conducted on demand.

It needs to be stressed that our practical results rest upon the analysis of a single software system. As it is not possible to generalize from a single case, additional cases in different industrial settings should be studied in the future to validate our results. Nevertheless, in comparison to our theoretical work the conducted case study bears the advantage that it is based on a real-life software system.

Acknowledgment

We want to thank our cooperation partners from KISTERS AG for enabling us to perform the industrial case study presented in this paper.

References

[1] R. Kazman and L. Bass, "Toward deriving software architectures from quality attributes," Defense Technical Informtion Center (DTIC), Tech. Rep., 1994.

[2] L. Chung and J. C. S. do Prado Leite, "On Non-Functional Requirements in Software Engineering." Springer Berlin Heidelberg, 2009, pp. 363–379.

[3] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 2nd ed. Boston, MA, USA: Addison-Wesley Professional, 2003.

[4] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[5] A. L. Wolf and D. E. Perry, "Foundations for the Study of Software Architecture," *ACM SIGSOFT Software Engineering Notes*, vol. 17, no. 4, pp. 40–52, 1992.

[6] J. Knodel and D. Popescu, "A Comparison of Static Architecture Compliance Checking Approaches," in *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA)*. IEEE, jan 2007.

[7] G. C. Murphy, D. Notkin, and K. Sullivan, "Software Reflexion Models: Bridging the Gap Between Source and High-level Models," *ACM SIGSOFT Software Engineering Notes*, vol. 20, no. 4, pp. 18–28, oct 1995.

[8] "hello2morrow - Sonargraph Architect." [Online]. Available: https://www.hello2morrow.com/products/sonargraph/architect9

[9] "Structure101 Software Architecture Development Environment (ADE)." [Online]. Available: http://structure101.com/

[10] L. J. Pruijt, C. Köppe, J. M. van der Werf, and S. Brinkkemper, "HUSACCT: Architecture Compliance Checking with Rich Sets of Module and Rule Types," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering (ASE)*. ACM Press, 2014, pp. 851–854.

[11] A. Caracciolo, M. F. Lungu, and O. Nierstrasz, "How Do Software Architects Specify and Validate Quality Requirements?" Springer, Cham, 2014, pp. 374–389.

[12] I. Melo, G. Santos, D. D. Serey, and M. T. Valente, "Perceptions of 395 Developers on Software Architecture's Documentation and Conformance," in *2016 X Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)*. IEEE, sep 2016, pp. 81–90.

[13] D. Garlan, S. Cheng, A. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure," *Computer*, vol. 37, no. 10, pp. 46–54, oct 2004.

[14] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and Hong Yan, "Discovering Architectures from Running Systems," *IEEE Transactions on Software Engineering*, vol. 32, no. 7, pp. 454–466, jul 2006.

[15] A. Nicolaescu, H. Lichter, A. Göringer, P. Alexander, and D. Le, "The ARAMIS Workbench for Monitoring, Analysis and Visualization of Architectures based on Run-time Interactions," in *Proceedings of the 2015 European Conference on Software Architecture Workshops (ECSAW)*. ACM Press, 2015, pp. 57:1–57:7.

[16] S. Herold, M. English, J. Buckley, S. Counsell, and M. O. Cinneide, "Detection of Violation Causes in Reflexion Models," in *Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering - SANER '15*. IEEE, mar 2015, pp. 565–569.

[17] "Dynatrace - Digital Performance & Application Performance Monitoring." [Online]. Available: https://www.dynatrace.com/

[18] B. Thoms, "The concept of dynamic analysis," *ACM SIGSOFT Software Engineering Notes*, vol. 24, no. 6, pp. 216–234, nov 1999.

[19] H. Yan, D. Garlan, B. Schmerl, J. Aldrich, and R. Kazman, "DiscoTect: A System for Discovering Architectures from Running Systems," in *Proceedings of the 26th International Conference on Software Engineering (ICSE)*. IEEE, 2004, pp. 470–479.

[20] D. Ganesan, T. Keuler, and Y. Nishimura, "Architecture Compliance Checking at Runtime: An Industry Experience Report," in *Proceedings of the Eighth International Conference on Quality Software (QSIC)*. IEEE, aug 2008, pp. 347–356.

[21] D. Le, A. Nicolaescu, and H. Lichter, "Adapting Heterogeneous ADLs for Software Architecture Reconstruction Tools," in *Proceedings of the 10th International Conference on Software Engineering Advances (ICSEA)*. Barcelona, Spain: IARIA XPS Press, 2015, pp. 52–55.

[22] S. Ducasse and D. Pollet, "Software Architecture Reconstruction: A Process-Oriented Taxonomy," *IEEE Transactions on Software Engineering*, vol. 35, no. 4, pp. 573–591, 2009.

[23] L. Passos, R. Terra, M. T. Valente, R. Diniz, and N. das Chagas Mendonca, "Static Architecture-Conformance Checking: An Illustrative Overview," *IEEE Software*, vol. 27, no. 5, pp. 82–89, sep 2010.

[24] J. Knodel, D. Muthig, U. Haury, and G. Meier, "Architecture Compliance Checking - Experiences from Successful Technology Transfer to Industry," in *Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, apr 2008, pp. 43–52.

[25] J. Rosik, A. Le Gear, J. Buckley, and M. Ali Babar, "An Industrial Case Study of Architecture Conformance," in *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ACM Press, 2008, pp. 80–89.

[26] L. D. Silva, "Towards Controlling Software Architecture Erosion Through Runtime Conformance Monitoring," Ph.D. dissertation, University of St Andrews, 2014.