Universität Stuttgart

Understanding Behavior of Third-Party Components through Explanation and Configurable Mapping Models

Von der Fakultät für Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Alireza Hakamian

aus Teheran

Hauptberichter:	Prof. Dr.-Ing Steffen Becker
Mitberichter:	Univ.-Prof. Dr.Uwe Zdun
Tag der mündlichen Prüfung:	15.Mar.2024

Institute of Software Engineering

2024

Abstract

Context. Most of the code executed in modern software systems is developed and maintained outside the organization. Understanding whether or not third-party components behave as intended requires expertise, which is rare and costly. Behavioral conformance checking (BCC) aims to identify deviations and conformances between system design and implementation by establishing relations between elements of design and implementation models.

Challenge. Previous studies have inadequately addressed the role of application designers who employ various third-party components. Firstly, existing methods do not evaluate how these methods enhance understanding of system behavior. Secondly, state variable mapping functions are implemented into the third-party components' codebase or as separate scripts, increasing the effort required to understand and modify mapping functions.

Objective. This dissertation aims to assist application designers in (1) simplifying BCC for different third-party components and (2) understanding the behavior of third-party components. Our method aligns more with design science methodology that aims to address research questions by developing and evaluating IT artifacts.

Results. To that end, we propose IMproved UNderstanding bEhavior (Imune), a tool-supported method. The solution in this dissertation ex-

tends the BCC process in two regards. First is the explanation method that assists the understanding of intended behavior and conformance results. Second is the configurable state variable mapping model promoting the separation of concerns and simplifying BCC's application to different third-party components. This dissertation validates the Imune method in (1) correctly identifying conformance and deviations and (2) understandability. Concerning understandability, we run a comparative user study with code inspection — code reviewing. The results demonstrate that Imune improves behavior understanding of complex third-party code implementations compared to code inspection.

Zusammenfassung

Context. Die meisten im modernen Softwaresystem ausgeführten Codes werden außerhalb der Organisation entwickelt und gewartet. Zu verstehen, ob Drittanbieterkomponenten wie beabsichtigt funktionieren, erfordert Fachkenntnisse, die selten und kostspielig sind. Das Überprüfen der Verhaltenskonformität (behavioral conformance checking) zielt darauf ab, Abweichungen und Übereinstimmungen zwischen dem Systemdesign und der Implementierung zu identifizieren, indem Beziehungen zwischen Elementen von Design- und Implementierungsmodellen hergestellt werden.

Herausforderung. Frühere Studien haben die Rolle von Anwendungsdesignern, die verschiedene Drittanbieterkomponenten einsetzen, unzureichend behandelt. Erstens vorhandene Methoden bewerten nicht, wie diese Methoden das Verständnis des Systemverhaltens verbessern. Zweitens werden Zustandsvariablenzuordnungsfunktionen in den Codebasis der Drittanbieterkomponenten oder als separate Skripte implementiert, was den Aufwand erhöht, um die Zuordnungsfunktionen zu verstehen und zu ändern.

Ziel. Diese Dissertation zielt darauf ab, Anwendungsdesignern bei (1) der Vereinfachung von behavioral conformance checking für verschiedene Drittanbieterkomponenten und (2) dem Verständnis des Verhaltens von Drittanbieterkomponenten zu helfen. Unsere Methode orientiert sich stärker an der Designwissenschaftsmethodik, die darauf abzielt, Forschungsfragen durch

die Entwicklung und Bewertung von IT-Artefakten zu beantworten.

Ergebnisse. Zu diesem Zweck schlagen wir IMproved UNderstanding bEhavior (Imune) vor, eine werkzeuggestützte Methode. Die Lösung in dieser Dissertation erweitert den behavioral conformance checking Prozess in zwei Hinsichten. Erstens ist die Erklärungsmethode, die das Verständnis des beabsichtigten Verhaltens und der Konformitätsergebnisse unterstützt. Zweitens ist das konfigurierbare Zustandsvariablenzuordnungsmodell, das die Trennung von Anliegen fördert und die Anwendung von behavioral conformance checking auf verschiedene Drittanbieterkomponenten vereinfacht.

Diese Dissertation validiert die Imune-Methode hinsichtlich (1) der korrekten Identifizierung von Konformitäten und Abweichungen und (2) der Verständlichkeit. In Bezug auf die Verständlichkeit führen wir eine vergleichende Benutzerstudie mit Codeinspektion. Die Ergebnisse zeigen, dass Imune das Verständnis des Verhaltens komplexer Implementierungen von Drittanbietercodes im Vergleich zur Codeinspektion verbessert.

Acknowledgements

I have dedicated over five years to learning architecture-based quality assurance. During the initial three years, I learned how to describe software architectures using formal languages like Petri nets and TLA+ alongside domain-specific languages such as the Palladio Component Model. The remaining time was devoted to engineering solutions for monitoring architecture descriptions at system runtime.

I would like to express my gratitude to those who played pivotal roles in shaping this dissertation. Steffen Becker taught me domain modeling and guidance and encouragement throughout my PhD journey. André van Hoorn consistently provided insightful feedback. He influenced me in identifying a research gap. His thorough review of the manuscript also helped catch many errors. Constant discussions with my PhD companion, Floriment Klinaku, influenced the foundational, solution, and validation chapters. Sebastian Frank and Sarah Stieß made helpful comments after test runs to validate this work.

Alireza Hakamian
Hamburg
02 May 2024

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1. Context

Today's software systems aim for zero downtime and faster release cycles [Dav19]. To realize those requirements, software engineering practices have introduced platforms to support development and operation. A notable example is the Java 2 Platform, Enterprise Edition (J2EE), first released in 2000 [Ora23]. Platforms offer two immediate benefits: reduced development costs and improved Quality of Service (QoS). They provide pre-built functionalities, saving developers time and effort. Additionally, platforms often enforce architectural patterns, such as J2EE's support for the multi-tier architectural style, enhancing QoS.

In the past decade, practices for developing and operating software systems have evolved, prompting the development of new platforms. Previously, developers and operators had to deal with infrastructure-centric primitives like *compute*, *storage*, and *network* to deploy and monitor system behavior. However, modern platforms like Kubernetes [IH19] have introduced new primitives such as containers, elevating applications to a first-class entity, and delegating underlying *compute*, *storage*, and *network* management to

infrastructure providers.

1.1.1. Background Information

To achieve the objectives of zero downtime and rapid release cycles, modern platforms adopt various architectural patterns [Dav19]. These patterns, including eventual consistency (for data or entire system state), redundancy of request and service instances, dynamic service discovery, configuration management, and Circuit Breaker, are prescribed to ensure system reliability amidst continual changes within the execution context—a concept known as resilience in the literature [Lap08]. These changes encompass frequent code releases, infrastructure outages, potential crash-failure modes in the underlying platform, and the relocation of application service instances.

Third-party designers or platforms often implement architectural patterns that application designers integrate into their software systems to enhance quality requirements. One such pattern is the Circuit Breaker pattern [Nyg18], which aims to improve system reliability. In a dynamically changing execution context, certain services may become temporarily unable to process requests due to factors like high load or network outages. The Circuit Breaker ensures that such services, which cannot handle requests, are not further degraded. If the Circuit Breaker fails to operate as intended at runtime— for instance, if it does not transition from *CLOSED* to *OPEN* and eventually to *HALF-OPEN*—the system's overall reliability is compromised. Therefore, it's crucial to verify whether third-party components behave as intended, constituting a critical operational activity.

Third-party components typically come with (1) a set of already defined metrics, which serve as indicators for operators to check what is happening in the system runtime execution, and (2) documentation that describes the design, metrics, and configuration parameters using natural language. However, interpreting monitored data and documentation to ensure third-party subsystems behave as intended demands expertise, often scarce and expensive. A possible solution is to create models for third-party implementations' intended and actual behavior and compare them to determine

whether actual behavior deviates from the intended. In that regard, model-based reliability and availability engineering (MBRAE) [TB17] practice aims to represent the intended behavior of software systems with changes foreseen within the execution context. By employing behavioral models, such as Markov chains [BGMT06], or simulation models [BGMT06], engineers predict that the described behavior satisfies system properties related to reliability and availability requirements. MBRAE is often associated with design-time analysis where the system is not in the implementation phase.

Reverse engineering (RE) is a practice that enables software engineers to extract the system's actual behavior from, e.g., implementation artifacts such as source code, executable binary, or observable data generated at runtime. RE originated in software maintenance (*IEEE-1219* standard). According to *IEEE-1219-1998*, reverse engineering is "the process of extracting software system information (including documentation) from source code." Examples of information extraction in RE encompass inferring (1) behavioral models [LMP08] and (2) properties [EPG+07] from the running system (i.e., binary code). Integration of MBRAE and RE practices leads to the development of two research areas: runtime verification (RV) [FHR13] and behavioral conformance checking (BCC) [AGR11; SAG+06].

1.1.2. Transition to the Challenge

RV aims to check if the system execution at runtime satisfies a set of specified system or component-level properties.

The RV process begins with engineers specifying system properties, typically employing temporal languages such as Linear Temporal Logic (LTL) [MJG+12]. Following this, the source code or binary undergoes instrumentation, achieved either through automatic generation from property specifications or manual addition of instrumentation code. This instrumentation facilitates monitoring the system's runtime behavior, enabling subsequent analysis to determine whether observed behavior aligns with the specified properties.

The BCC process offers a systematic approach to identifying conformances

and deviations between a system's intended and implemented behavior. One possible approach to achieving this is for engineers to develop a formal system design specification, extract a behavioral model from system runtime and compare the two. One commonly used formalism for the specification is transition systems [BK08]. Informally, transition systems are defined over a finite set of states, where each state represents assignments of values to a set of variables. Consequently, a deviation is, for instance, when a state present in the design model is absent in the observed system behavioral model. Mapping state variables from specifications to their implementations is a prerequisite for the comparison. This mapping process entails establishing correspondences between variables in a specification and their implementation using a set of mapping functions. Consider an example in the context of a Circuit Breaker software system. In the formal specification, the state variable *circuitBreaker* takes either of the following values

- *closed.* The value represents a normal operation of the Circuit Breaker, allowing requests to go to the target service.
- *open.* The value represents that no request is permitted to go to the target service.
- *half-open.* The value shows a test state where a limited number of requests are allowed to test the target service's health.

After implementing and executing the Circuit Breaker system, here are the observed values for the implemented variable *circuitBreaker*.

- *closed.* The value represents a normal operation of the Circuit Breaker, allowing requests to go to the target service.
- *open.* The value represents that no request is permitted to go to the target service. Moreover, after a few seconds, test requests check the target service's health.

A state variable mapping allows for creating a correspondence between the specification variables and their implementation, providing insights into the system's behavior and identifying any deviation. After mapping,

the comparison from the example must indicate that the *half-open* state is missing.

1.2. Challenge

Applying BCC or RV for third-party components by application designers' role provides confidence in the correct behavior of the third-party components integrated into the overall system. Investigating a subset of existing methods (discussed in related work) under the RV and the BCC reveals that the application designers' role has not been adequately addressed.

First, state variable mapping, as defined in previous works such as Schvimer et al., [SDH20], involves encoding mapping functions directly into the implementation codebase or hard-coding them into separate scripts. There are two main drawbacks to this. Firstly, it lacks flexibility in adapting the mapping functions to accommodate changes in design specifications or the implementation of third-party components. Adapting involves modifying existing mapping functions or adding new ones, which can be time-consuming, error-prone, and may introduce unintended side effects. Secondly, the less explicit representation of mapping functions within the codebase makes them challenging to read and understand. Lacking flexibility and explicitness can result in significant time and effort being invested in maintaining mapping functions for different third-party components, ultimately slowing down the adoption of behavioral conformance checking (BCC) applications.

The second is application designers' understanding of the behavior of third-party components after applying BCC or RV methods. Application designers are not part of the design specification and implementation of the third-party components. Hence, understanding third-party components' behavior is challenging for application designers using third-party components. Such understanding is required for further troubleshooting or communication with third-party designers. The following lists why understanding the outputs, including as-Intended behavior and deviations between as-Intended and observed behavior during runtime execution, is essential.

- **Effective collaboration.** Reducing misunderstandings and promoting efficient cooperation between application designers, such as developers and operators.
- **Knowledge transfer.** Different individuals or teams may maintain third-party components within the application over time. Understandability ensures that the knowledge about the BCC process is transferred effectively from one person or team to another.
- **Acceptance.** Understandability is crucial in adopting and accepting a BCC method.

The following research questions will guide the investigation and provide a framework for exploring the key aspects of the study.

- *Research Question 1.* How to extend existing BCC methods to incorporate the role of application designers for effective conformance checking of third-party components?

  Hypothesis. The development of extensions to address the application designer role simplifies behavioral conformance checking of different third-party components within the software system.

  Additional Context. Simplification refers to flexibility and explicitness in the state variable mapping method.

- *Research Question 2.* How does applying the method enhance the understanding of third-party code behavior compared to established practices, such as code inspection — code reviewing?

  Hypothesis. The application of the method improves the understanding of third-party components' behavior compared to established methods such as code inspection.

1.3. Objectives and Solution Overview

This dissertation aims to aid application designers, who are end-users of third-party components, by (1) simplifying BCC for different third-party

components within the system and (2) enhancing understanding of their behavior. Simplification is achieved by introducing a state variable mapping model. To that end, we propose **IM**proved **UN**derstanding b**E**havior (Imune). Figure 1.1 gives an overview of how Imune benefits existing tasks in the operation and maintenance processes [ISO15]. Monitoring ensures that the

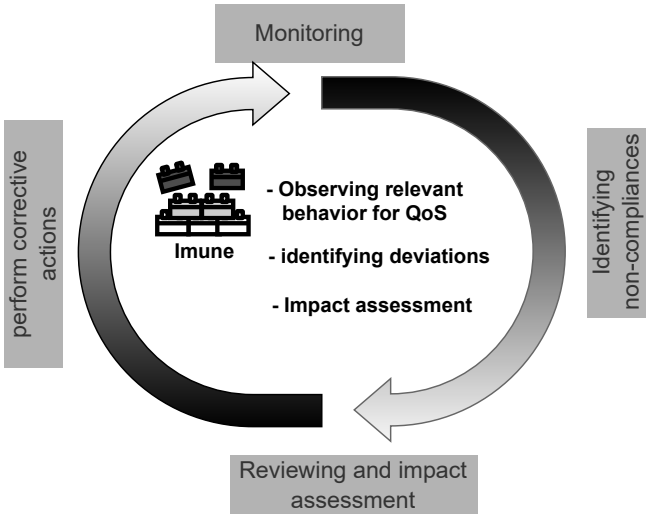

Figure 1.1.: Imune used in operation and maintenance process

system operates according to its agreed-upon Quality of Service (QoS), with Imune guiding the extraction of relevant behavior for QoS alignment. Detecting deviations has been discussed before as essential for quality-assured software systems using third-party code. During the review and impact assessment phase, application designers should determine the consequences of deviations in system properties. The reviewing phase outcome enables application designers to perform corrective actions when needed. For instance, in the Circuit Breaker example, the observed behavior from an implementation [Hys12] shows that the state *half-open* is not considered in the implementation. One approach is communicating the identified deviations with the third-party component designers to determine the consequences.

The missing state most likely suggests (1) an incomplete implementation or (2) an implementation that merges the *open* and *half-open* states into one. Both scenarios have different consequences on the system properties. The first one indicates that the implementation does not satisfy its design specification. Whereas the second indicates system properties and design specification must be revised to incorporate no *half-open* state. Nevertheless, impact assessment may require modifying design specifications, which needs collaboration with formal method experts. While Imune guides impact assessment, the task is not automated.

Figure 1.2 illustrates a general conformance checking process model. The activities are (1) extracting as-implemented architecture, (2) specification of as-designed architecture, (3) comparison, (4) verification of violations, (5) suggesting and changes.

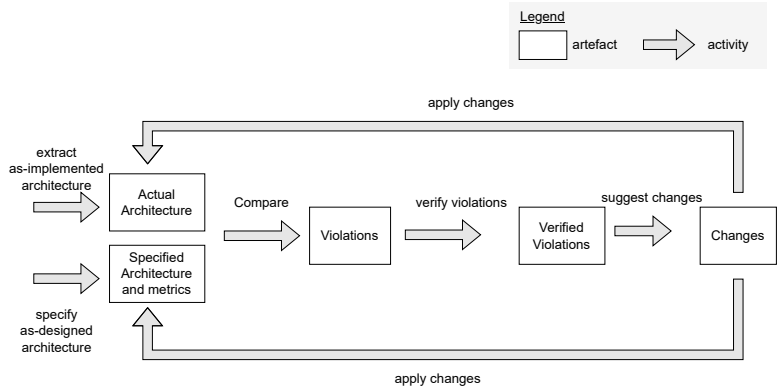

Figure 1.2.: Conformance checking process (based on [TCL04])

The solution in this dissertation extends the BCC process in two regards. First is the explanation activity that assists the understanding of intended behavior and conformance results. Second is the state variable mapping configuration model facilitating BCC's application to different third-party components. Both extensions are visible in Figure 1.3.

This dissertation claims that *The Imune method improves behavior un-*

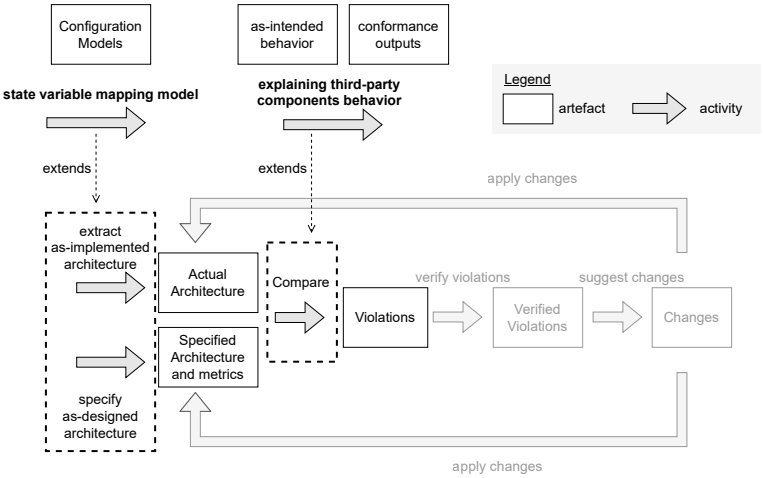

Figure 1.3.: Using the Imune method for behavior conformance checking of third-party components

derstanding and simplifies conducting behavioral conformance checking for different third-party components within the system. The following analyzes the claim in more depth.

- **Behavior understanding.** Imune employs an explanation of as-intended behavior and conformance results. We validate this claim through a comparative user study with the code inspection [Fag02; Fag76], a method to statically validate implementations code against requirements specification or coding standards, among others.
- **Simplifies conducting conformance checking.** The approach introduces state variable mapping configuration templates associated with each third-party component. The configuration model allows flexibility in adapting the mapping method without modifying the codebase that involves implementation details irrelevant to the mapping. Additionally, the configuration model positively contributes to the explicit documentation of the mapping method. Overall, using a configuration

model simplifies the complexity of the mapping method within the BCC process.

- **Third-Party components.** The Imune method applies to different third-party components within the software system in a unified manner.

1.4. Method

According to Wohlin et al. [WA15], empirical research in software engineering can be followed in three steps: (1) challenge identification and research question formulation. (2) study design and (3) interpreting results and findings of research. Conducting research in the context of this PhD aligns more with the design science methodology [HMPR04] in software engineering, where the goal is to address the research questions by developing and evaluating IT artifacts, such as models or methods for real-world challenges. For us, applying the design science methodology involves (1) clearly describing research challenges, objectives and requirements for a possible solution, (2) developing the new method and tooling that supports defined objectives and requirements, (3) evaluating the method's effectiveness empirically by performing experiments.

From author of this dissertation's point of view, arriving at a challenge that can be tackled within PhD scope requires a few research iterations. Each research has its decision-making, such as research objective or data collection methods. Below is an overview of the research done for challenge clarification during this PhD. More detail is in Section 3.

Often, from the beginning, the challenge and research questions are very broad and often unknown. For example, this dissertation's author initially investigated emerging quality attributes, including resilience, survivability, and elasticity. The guiding questions are why they have emerged and what are the differences between them. The research finding is that resilience has emerged to advocate development and operation practices for ensuring that systems never face unacceptable QoS degradation. The results from the previous research activity are the next step, along with the guiding question

of why software systems face unacceptable QoS degradation. The findings led to problems and research questions that this dissertation aims to tackle.

1.5. Contributions

A list of the main contributions of this PhD dissertation is as follows.

1. **Imune method.** Proposing Imune as a systematic approach for third-party components behavioral conformance checking, which uses the state variable mapping model and explanation of as intended behavior and conformance checking results. The method is effective in terms of identifying deviations and understandability by end-users of third-party components.

   Imune process model. The Imune process model describes the steps required for formal method experts and third-party library designers to define the state variable mapping. Moreover, the process describes the steps for end-users of third-party components to detect deviations and understand the outputs.

   Imune tooling. Imune tooling supports Imune process model implementation

2. **Imune method evaluation.** We evaluate the Imune effectiveness in correctness and understandability, following experiment design in software engineering by Wohlin [WRH+12]. The supplementary materials, including the tool, are publicly available [Hak24]. Correctness refers to the ability to detect deviations. Understandability refers to how well Imune assists in understanding the as-intended and as-implemented behavior of the third-party components. Such understanding is often a precondition for re-configuration or communication with third-party code designers concerning deviations. For both criteria, correctness, and understandability, we conduct an experiment using two software projects Producer-Consumer and ReadWrite patterns. We evaluate correctness by using the two software projects. For understandability,

we conduct a between-subject user study with 10 participants with various educational levels, including PhD, Master, and upper level ($\geq$ 6 semester) Bachelor students. The purpose is to perform a comparative analysis of Imune and an established method for behavioral conformance checking, code inspection. Answers to a set of quizzes are means to assess understandability scores. Performing comparative analysis provides two valuable benefits. First, comparison with a well-established method for behavior conformance checking validates the benefit of Imune in practice. Second, the comparison provides a benchmark for assessing BCC effectiveness for third-party code.

1.6. Running Example

This section provides a short description of Circuit Breaker, serving as a running example for the entire dissertation. The reason is that some concepts, such as system properties and verification, become more precise when applied to an example.

Based on the Circuit Breaker pattern description from [Nyg18], the calling target service will likely fail in a distributed environment, e.g., due to network issues. The resources of the caller get wasted for a non-responding service, which impacts the service's quality. The problem is known as cascading failure. The Circuit Breaker pattern solves this problem by disallowing further calls to the target service if a certain failure threshold is reached. The expected states of the Circuit Breaker are either (1) closed, (2) open, or (3) half-open. In a closed state, all requests can go to the target service. In an open state, no requests are allowed to go through the Circuit Breaker and reach the target service. Whereas, in half-open, a limited number of calls are allowed to check the health of the target service.

1.7. Structure

Chapter 2 starts with a taxonomy classifying relevant concepts for BCC and our strategy in searching the literature to identify relevant works. The concepts are related to MBRAE, RE, and architecture-based conformance checking. Moreover, starting from Section 2.3, there is an in-depth discussion of the conformance definition in the literature. In particular, definitions for conformance from formal methods and software engineering perspectives. In the end, the section provides (1) a list of requirements for BCC to address the needs of the application designer role and (2) judging identified deficiencies based on listed requirements. **Chapter 3** details various iterations and related research design decisions to identify research challenges in the scope of this PhD. We explain the Imune method in the **Chapter 4**. In addition, Section 4.5 discusses existing assumptions and limitations of Imune. Validation of the Imune method is discussed in **Chapter 5**. **Chapter 6** summarizes the entire works done in this dissertation and discusses open questions that provide insights for future works.

CHAPTER 2

Foundations and Related Work

This chapter introduces key concepts in two practices: (1) model-based reliability and availability engineering (MBRAE) and (2) reverse engineering (RE). Moreover, the chapter provides an in-depth discussion of related research areas on behavioral conformance checking (BCC). Additionally, we provide a taxonomy to visualize the concepts associated with the two practices within BCC. The taxonomy illustrates the concepts explained in this chapter. The end of this introductory text explains our strategy for reviewing the literature and collecting state-of-the-art methods related to this dissertation's contributions.

The MBRAE practice ensures the development of software systems that meet its reliability or availability requirements. In MBRAE, the notion of a formal design model is central. Software engineers build a model of the to-be-implemented software systems considering foreseeable environmental changes such as software or hardware failures. Depending on the roles involved in the development team, different modeling languages exist, rang-

ing from less intuitive for many software engineers, such as classical logic and set theory, to more intuitive languages, such as Petri Nets and Unified Modeling Language (UML). The analysis results predict whether the system fulfills its reliability and availability requirements. Depending on the model, the analysis can be (1) rigorous in proofing the absence of error in the software design, which is often associated with exhaustive verification, or (2) approximately answering whether or not software designs satisfy quality requirements, which often is associated to simulation and asymptotic assessment.

Software maintenance is defined by *IEEE-1219-1998* as modification of software systems after delivery to perform activities, including fault correction, quality requirements improvement, or adapting the software to changing execution context. RE is part of software maintenance, extracting high-level information required for tools or humans for maintenance activities [CPC11]. The high-level information refers to software architecture documentation or trace links between software artifacts. According to *IEEE-1219-1998*, RE does not change the source code but provides a different representation of the software system aiming to address software engineering needs, such as recovering software architecture and program documentation. The input for building different representations includes but is not limited to design documentation, implementation source/binary code, and runtime observable from system execution.

The taxonomy, part one in Figure 2.1 and part two in Figure 2.2, visualizes associated concepts of the two practices within BCC.

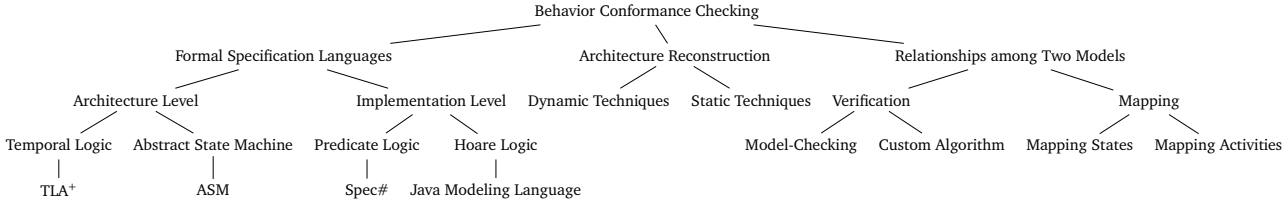

Figure 2.1.: Overview of relevant concepts for behavior conformance checking - Part 1

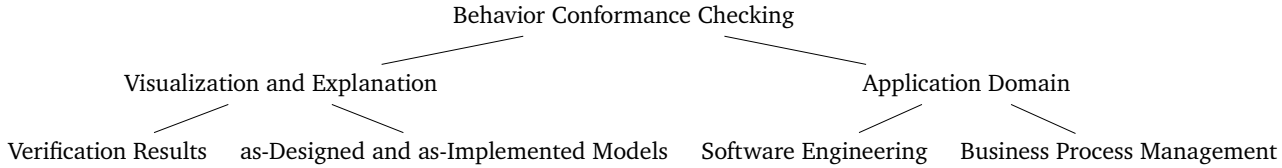

Figure 2.2.: Overview of relevant concepts for behavior conformance checking - Part 2

The state-of-the-art methods related to the contributions of this dissertation are collected based on a structured literature review. For us, the primary purpose behind reviewing the literature for this dissertation is to reach an acceptable level of confidence that claimed contributions are indeed novel. The Structured literature review is rigor similar to the systematic literature review proposed by Kitchenham et al., [KC07], with less detailed analysis such as statistical analysis, e.g., number of publications per year, meta-analysis, e.g., gaps and research directions.

1. **Objectives.** To Identify relevant BCC research areas and summarize state-of-the-art methods within each area.
2. **Inclusion and exclusion criteria.** For us, the primary inclusion criteria are that (1) papers' definition of conformance aligns with the definition in Section 2.3.1, and (2) papers can be judged as related to architecture-based conformance checking.
3. **Search Strategy.** The strategy is to collect papers by searching in databases including *Google Scholar* and computer science bibliography, i.e., *Dblp*. The main keywords used for searching are *conformance/consistency/equivalence*[1] *checking*, and *implementation relations* combined with *behavior-based*, and *architecture-based*. Additionally, we use the keyword *TLA*$^+$ to extract language-specific related works. Moreover, studies have been added to the collection based on the visibility of the author of this dissertation. The visibility is related to the conferences, including *ECSA* and *ICSA*.
4. **Data Extraction and Analysis.** The proposed approach, evaluation of key findings, and limitations are essential for us. Regarding the analysis, existing works are classified and discussed starting from Section 2.3.1.

We achieve our objective of identifying relevant research areas for BCC. They are following.

[1] Sometimes, the term equivalence appears in the literature associated with formal method studies, while consistency has appeared in software engineering works.

- **Program verification.** It combines symbolic and concrete execution of programs, including source or binary code. The primary goal is to detect errors at the program level. An example approach for verification of Java programs is *Java PathFinder* [HP00] and for C programs is *BLAST* [BHJM07].
- **Runtime Verification.** Approaches within this category mainly concern the current execution path and not all possible inputs and execution paths, unlike program verification. Design-by-contract approaches such as JML [BCC+05] and Monitor-oriented programming [MJG+12] are example approaches.
- **Architecture-based conformance checking.**Unlike the above two research areas that check whether implementation satisfies system-level properties, architecture-based conformance checking approaches compare two system models at different abstractions. What is precisely meant by comparison depends on the model, described later in Section 2.3. There exist two categories for ensuring architecture conformance. They are conformance by construction and conformance by extraction. Conformance by extraction is more relevant for us as the implementations of third-party components already exist. An example approach for conformance by extraction is reflexion modeling [MNS01].

2.1. Formal Models and System Properties

Formal Modeling is a central activity in MBRAE. Software engineers use formal models to describe software systems' behavior and their execution context and desired properties [TB17]. The language for specifying the system has mathematically defined syntax and semantics [CW96] such as regular algebra, transitions systems, state machine, Markov models, Petri Nets, and Queuing Networks. Specifying properties is possible using provided constructs in the modeling language, or logical languages such as Linear temporal logic (LTL).

Software systems operate within an execution context, necessitating consideration of this context in system modeling for usefulness. The system context, encompassing factors like user behavior or access to required hardware capacity, is crucial for defining and verifying system properties [Poh10].

Model-based reliability engineering guides engineers in designing systems by considering different failure modes and recovery actions on system properties. Fault Tree Analysis (FTA) and Failure Mode and Effect Analysis (FMEA) are examples of practices that can help us better understand failure modes.

Modeling languages facilitate the creation of executable artifacts representing systems' designs, including their execution context and related properties. However, the designs are typically abstract to enable efficient property checking. Thus, in Model-Based Reliability Engineering (MBRAE), incremental refinement from abstract to implementation models is vital, often resulting in executable code, like *Java*. The Temporal Logic of Actions (TLA) [Lam94]is a modeling language based on transition system formalism, allowing engineers to use temporal logic to specify system designs.

The following explains in detail transition systems (TSs) as the formalism and TLA as the language based on TS for modeling (1) software systems' design, (2) their execution context, and (3) desired system properties.

2.1.1. Transition Systems and TLA$^+$

Pragmatism is one characteristic of models. For example, checking whether or not software systems reach deadlocks is a possible pragmatism of why engineers create models. Choice of formalisms, modeling languages, and tools often depends on system types, properties, and environment in which the system is being executed [CHV18]. Various system types, such as probabilistic, concurrent, and timed, can be modeled using Markov chains, Petri Nets, and timed automata formalisms. Similarly, there are different classes of system properties, such as real-time properties, which can be specified using real-time temporal logic or checking deadlock, for which a simple propositional logic is enough.

Transition systems are often the formalism choice for describing the behavior of software systems in discrete steps. The representation of transition systems is the same as a graph where nodes are states and edges represent transitions, i.e., state changes. A state represents assignments of values to a set of variables, and a transition shows an event that assigns new values to a subset of variables. The assignment of values to the variables forms a new state.

A transition system TS is a tuple $(S, Act, \rightarrow, I, AP, L)$ [BK08].

- S is a set of states,
- Act is a set of actions that the system takes,
- $\longrightarrow \subseteq S \times Act \times S$ is a transition relation,
- I is a set of initial states,
- AP is a set of atomic propositions, and finally
- $L : S \rightarrow 2^{AP}$ is a labeling function, and 2^{AP} is a powerset, i.e., a set of all subsets of atomic propositions.

States S represent assigning values to a finite set of typed variables $V = \{v_1, v_2, ..., v_n\}$, which ranges over discrete domains such as Boolean or integers. Hence, a state is an element in $\prod_{v_i \in V} D_{v_i}$. The transition system starts in an initial state $s_0 \in I$ and moves to the next state by action a s' ($s_0 \xrightarrow{a} s'$).

When more than one transition is enabled from a state, the choice of what transition to take is non-deterministic. Similarly, when more than one initial state exists, the choice of what state to pick first is non-deterministic.

The labeling function L relates a set of atomic propositions to any state s. Atomic propositions characterize a set of states through possible variable values in them. For example, for variable x ranging over integers, the atomic formula $x = 1$ characterizes states where variable x equals 1 is true in them. A set of direct *a-successors* for a state $s \in S$ and $a \in Act$ is defined as $Next(s,a) = \{s' \in S \mid s \xrightarrow{a} s'\}$, $Next(s) = \bigcup_{a \in Act} Next(s, a)$.

TLA [Lam94] is a modeling language based on transition system formalism, allowing engineers to use temporal logic to specify system design and properties. TLA [Lam94] support design and property specification in a single language. The primary purpose of TLA is the specification (properties and design) of distributed and concurrent systems. TLA is a type of transition system. TLA$^+$ [Lam02] is an extension to TLA, which makes the language suitable for writing modular specifications.

The following describes a simple hour clock example provided by Lamport in his book [Lam02] to show the language constructs in specifying actions and states.

The variable *hour* represents actual *hour*, which takes values $[1-12]$. A sequence of states for the hour clock is $hour = 1, hour = 2, \ldots$. The initial state specification of the system is $Init \triangleq hour \in (1..12)$, which means the system can start at any hour. After the initial state, the next action is *tick*, which specifies how the hour changes. The *hour* is: $Tick \triangleq hour' = IF\ hour \neq 12\ THEN\ hour + 1\ ELSE\ 1$. The formula is an action because it changes the value of the variable hour. The change is shown by using the $'$ symbol. The Formula $HClock \triangleq Init \wedge \Box Tick$, specifies the complete hour clock system. Formula $HClock$ is true if and only if $Init$ is true and $Tick$ is true for every step of the system behavior.

Consider a more complex example, the Circuit Breaker system. An execution context for the system is that multiple concurrent threads attempt to get permission from the Circuit Breaker to send requests. Hence, multiple threads are to be considered as the system execution context. Given the context, the Circuit Breaker switches between closed, open, and half-open states based on failure thresholds that the system records from calls to target service.

The following are the variables to model the given description.

calls is a function that maps OS threads $\{t1, t2, ..., t_n\}$ to the record of the form *[reply, permitted, permittedInState]*. The *reply* field is either *TRUE*, which means the service returns no error, *FALSE*, which means the service returns an error, or *NULL*, which means the thread has not sent any request yet. The *permitted* field shows whether (or not) the thread obtained permission to send a request. Finally, the *permittedInState* field tracks in which state the thread $t1$ obtained permission.

circuitBreaker takes either the *CLOSED*, *OPEN*, or *HALF* value to represent the current state of the Circuit Breaker.

recordedCalls is a sequence of recorded *TRUE*, or *FALSE* when the Circuit Breaker was in either the *OPEN* or *CLOSED* state.

Formula 2.1 shows possible initial states that the system can be.

$$\begin{aligned} Init \triangleq\ & calls = [\, t \in Threads \mapsto [\, reply \mapsto NULL, \\ & \qquad permitted \mapsto NULL, \\ & \qquad permittedInState \mapsto NULL\,] \\ & \wedge\ circuitBreaker \in \{CLOSED, HALF, OPENED\} \wedge\ recordedCalls = \langle\,\rangle \end{aligned} \tag{2.1}$$

2.1.2. Possible Behaviors: Executions, Paths and Traces

Contents of this section is based on [BK08].

Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system. A finite execution fragment σ of TS is a sequence of states and actions ending with a state:

$$\sigma = s_0 a_1 s_1 a_2 ... a_n s_n \mid s_i \xrightarrow{a_{i+1} s_{i+1}} \forall\ 0 \leq i < n,\ n \geq 0$$

n is length of execution fragment σ. An infinite execution fragment ρ of TS is defined as follows.

$$\rho = s_0 a_1 s_1 a_2 s_2 a_3 ... \mid s_i \xrightarrow{a_{i+1} s_{i+1}} \forall\ 0 \leq i$$

An execution fragment is called *initial* if $s_0 \in I$, in which I is the set of initial states. Moreover, an execution fragment is maximal if the execution fragment is either infinite or finite that ends in a *terminal state* where no transition is possible. An execution of transition system TS is initial and maximal. A state s is called reachable if there exists an initial and finite execution fragment that ends in s and starts in an initial state s_0.

Considering state-based execution, a transition system TS is deterministic if the number of initial states is at most one and $|Next(s) \cap \{s' \in S \mid L(s') = A\}|$ is at most one, which means there is maximum one outgoing transition leading to a state with Label A.

Ignoring the actions and focusing on the sequence of states result in paths. A finite path fragment $\hat{\pi}$ is a finite state sequence $s_0 s_1 ... s_n$ such that s_i is reachable from s_{i-1} for all $0 \leq i < n$, $n \geq 0$, $n \geq 0$. An infinite path fragment π is an infinite state sequence $s_0 s_1 s_2 ...$ such that s_i is reachable from s_{i-1} for all $i > 0$. A path of transition system TS is both initial and maximal.

In a path, states are not observable, but their atomic propositions are. Sequences of form $L(s_0), L(s_1), ...$ that is a set of valid atomic propositions along the path is called *traces*. More precisely, the trace of the infinite path fragment $\pi = \{s_0, s_1, ...\}$ is defined as $trace(\pi) = L(s_0), L(s_1), ...$ and the trace of finite path fragment $\hat{\pi} = \{s_0, s_1, ..., s_n\}$ is defined as $trace(\hat{\pi}) = L(s_0), L(s_1), ..., L(s_n)$. A trace of state s (*trace(s)*) is the trace of an infinite path fragment π, which starts in s. A finite trace of s is the trace of a finite path fragment that starts in s. *Traces(s)* defines a set of all traces of s, and *Traces(TS)* defines the set of traces of TS that starts in $s \in I$.

The Circuit Breaker is a transition system TS_{CB} with a finite number of states generated by the model-checking technique. Figure 2.3 shows a trace of the path fragment $\pi_{CB} = \{s_0, s_1, s_2\}$ is $trace(\pi) = L(s_0), L(s_1), L(s_2)$. Note that the path fragment is not maximal as the state s_2 is not terminal, and the sequence of the states is not infinite. The trace $trace(\pi)$ serves as an

example. The Circuit Breaker transition system generated by the TLC model-checker has infinite path fragments. More precisely, $Traces(TS_{CB})$ is defined as following where $Paths(s)$ are paths of the Circuit Breaker transition system such that paths are both initial and infinite.

$$\bigcup_{s \in I} Traces_(s) \quad and \quad Traces_(s) = trace(Paths_(s))$$

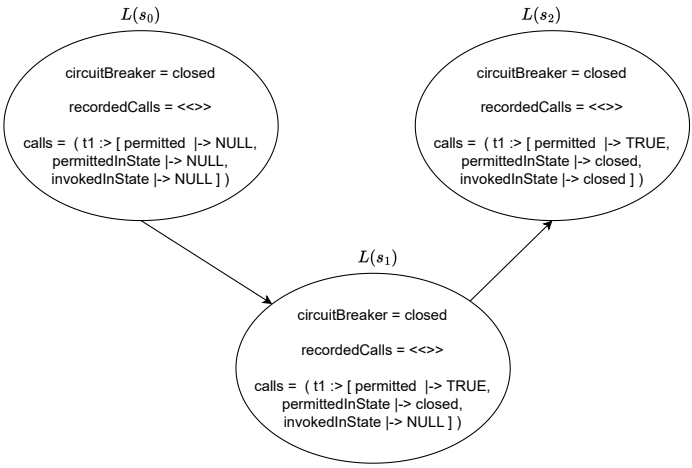

Figure 2.3.: Trace of the finite path fragment π_{CB}

2.1.3. Linear-Time Properties and Linear Temporal Logic

Linear-time (LT) properties describe traces that a transition system TS must exhibit [BK08]. LT properties are, in fact, requirements on the traces of TS. When LT property P is satisfied by the TS, it is shown by satisfaction relation $TS \models P$. TS satisfies P if and only if *Traces(TS)* $\subseteq$ P A state s satisfies P ($s \models P$) if and only if *Traces(s)* $\subseteq$ P.

From the perspective of Linear Temporal Logic (LTL), an execution is understood as a sequence of states. When non-determinism is involved, multiple paths become possible, meaning the system exhibits a range of potential behaviors. LTL expands upon propositional or predicate logic by incorporating modal operators [BK08].

In propositional logic, sentences are assigned Boolean values of either *True* or *False*, and there are five primary operators: conjunction (and) $\wedge$, disjunction (or) $\vee$, negation (not) $\neg$, implication (implies) $\Longrightarrow$, and equivalence (is equivalent to) $\equiv$. The definitions of these Boolean operators are provided in Table 2.1.

Operator	Formula	Interpretation
$\wedge$	$P \wedge Q$	equals true iff both P and Q equal true
$\vee$	$P \vee Q$	equals true iff both P or Q equal true
$\neg$	$\neg P$	equals false iff P equals true
$\Longrightarrow$	$P \Longrightarrow Q$	equals false iff P equals true and Q equals false
$\equiv$	$P \equiv Q$	equals true iff both equal false or both equal true

Table 2.1.: Proposition logic example

Predicate logic extends propositional logic with two additional quantifiers: *for all* ($\forall$) and *there exists* ($\exists$). For example, the formula $\forall n \in \text{Nat} : n+1 > n$ asserts that $n+1 > n$ is true for all members of the set of natural numbers. The formula $\exists n \in \text{Nat} : n^2 = 2$ asserts that there exists at least one number whose square is equal to 2, which is false.

LTL-formulas is based on (1) atomic propositions, (2) boolean operators conjunction $\wedge$ and negation $\neg$ and (3) two basic temporal modalities *next* $\bigcirc$ and *until* $\mathcal{U}$ [BK08]. The atomic proposition $a \in AP$ represents state label a in a transition system, which often are assertions about values of program variables [BK08]. Using conjunction and negation, other boolean operators can be derived [BK08].

The meaning of the two new operators is as follows [BK08]: Formula $\bigcirc \varphi$ holds at the current moment when Φ holds in the next step. Formula $\varphi_1 \mathcal{U} \varphi_2$ holds at the current moment if there is some future moment for which φ_2 holds, and φ_1 holds at all moments until that future moment.

The following grammar [BK08] describes the syntactic rule for constructing LTL formulas.

$$\varphi ::= \text{true} \mid \text{a} \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \bigcirc \varphi \mid \varphi_1 \,\mathcal{U}\, \varphi_2$$

The until operator allows deriving temporal modalities *sometimes in the future* (eventually) $\Diamond$ and *from now on forever* (always) $\Box$ [BK08]. By combining the two operators, new temporal modalities are obtained, which are *always eventually* $\Box \Diamond$ and *eventually forever* $\Diamond \Box$ [BK08]. The meaning of formulas are as follows [BK08]: Formula $\Box \Diamond a$ describe the path property that at any moment k there is a moment $m \geq k$ at which state a is visited. Formula $\Diamond \Box a$ describes from some moment k on only states a are visited. Figure 2.4 illustrates the semantics of temporal modalities introduced by LTL.

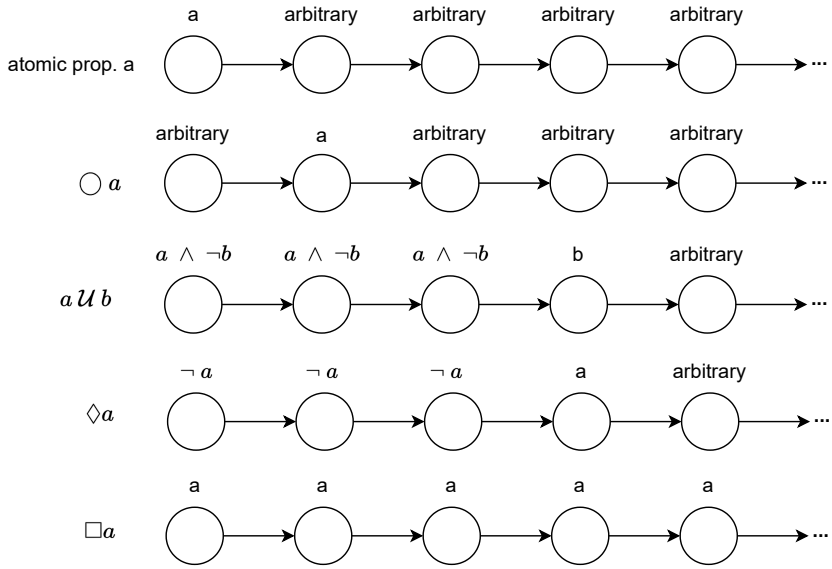

Figure 2.4.: Graphical description of temporal modalities [BK08].

Consider the Circuit Breaker example. The property

$$\Box\, \neg(\exists\ t \in Threads :\ calls[t].permittedInState = OPENED)$$

is an example of the $\Box$ temporal modality that says it is *always* never the case that q thread gain permission when Circuit Breaker is in the open state. Note that the $\Box$ can also be dropped without semantic changes.

2.1.4. Verification by Model-Checking

Verification tells whether or not software system models at any abstraction level, such as design, implementation, or observable, satisfy certain properties [BK08; CHV18]. One widely used verification technique in the formal methods community is model-checking. Clarke et al., [CHV18] explain that algorithms decide whether "a finite state transition structure is a model of a temporal-logic formula. Moreover, the decision procedure can produce diagnostic counterexamples when the formula is not true in the structure."

In contrast to model-checking techniques, which exhaustively search the entire state-space of the model and prove the absence of the bug, software testing can only show the presence of bugs and no guarantee of the absence of bugs.

2.1.5. Comparing Transition Systems: Refinement and Mapping

In the context of modeling languages such as TLA$^+$ [Abr10], or Event-B [Abr10], refinement is a systematic and rigorous software development approach, ensuring that the abstract specification is gradually transformed into a concrete implementation while preserving both the desired properties and behaviors. It provides a traceable path from high-level requirements to detailed and verified implementation. In detail following is the process.

- **Abstract Specification.** Software engineers specify what the system should do in the form of system properties and how the system within its execution context satisfies those properties. This step is abstract,

as the goal is to ignore unnecessary details irrelevant to property satisfaction. In the context of TLA$^+$, modeling involves specifying variables and actions.

- **Refinement Step.** In this step, engineers add more details to the abstract specification. Implementation is supposed to be a more detailed version of the abstract. These details do not exist in the abstract specification.
- **Mapping.** The mapping involves defining the relation between the abstract and implementation, which describes how the abstract concepts, such as variables and actions, are realized in the implementation. In transition systems, mapping is related to actions (or operations in more generic terms) and variables.
- **Verification.** Along each refinement step, the refined implementation can be formally verified to ensure that it preserves the properties and behaviors specified in the abstract specification. This verification process helps establish the correctness and consistency of the refinement process.
- **The loop.** The previous steps continue until the desired level of implementation is reached such that refined implementation is complete (i.e., it captures all the required details) and correct (i.e., it satisfies the properties specified in the abstract specification).

Next, let's explain the refinement process using the Circuit Breaker example. Formula 2.2 shows an abstract specification of Circuit Breaker, which only models the valid sequence of changes between states using the variable, *cbState*. Moreover, the change between states is triggered by events including *Success*, *Failure*, or *AttemptReset*, which are defined as constant parameters of the abstract model.

$$
\begin{aligned}
&CONSTANTS\ CLOSED,\ HALFOPEN,\ OPENED,\\
&\qquad\qquad Success,\ Failure,\ AttemptReset\\
&VARIABLES\ state\\
&Init \triangleq state = CLOSED\\
&Open(event) \triangleq state = CLOSED \land event = Failure \land state' = OPENED\\
&Close(event) \triangleq state = HALFOPEN \land event = Success \land state' = CLOSED\\
&FromOpenToHalf(event) \triangleq state = HALFOPEN \land event = Failure\\
&\qquad\qquad \land state' = OPENED\\
&HalfOpen(event) \triangleq state = OPENED \land event = AttemptReset\\
&\qquad\qquad \land state' = HALFOPEN\\
&Next \triangleq \exists e \in \{Success, Failure, AttemptReset\} : Close(e)\\
&\qquad \lor FromOpenToHalf(e) \lor HalfOpen(e) \lor Open(e)\\
&Spec \triangleq Init \land \Box[Next]_{state}\\
&ValidSequence \triangleq \Box[\sim(state = CLOSED \land state' = HALFOPEN)]_{state}
\end{aligned}
\tag{2.2}
$$

Formula 2.3 refines the abstract specification of Circuit Breaker developed with a module named *circuitBreakerAbstract*. The refined version of Circuit Breaker adds more details by modeling *timeout* aspect when the Circuit Breaker must stay for a certain amount of time in the *open* state.

In the verification activity during the refinement process, it's essential that the refined model satisfies both the properties and behaviors specified in the abstract model. Properties may not capture all aspects of the system's behavior. Behavior verification allows for a more detailed examination of how the refined model behaves under various conditions, uncovering potential

issues that may not be captured by property verification alone.

Regarding property verification, model-checking the refined model shows no violation of property. Concerning the behavior, by specifying *Model!Spec* as the temporal property, model-checking shows no violation meaning that the refined model mimics all possible behavior of the abstract model.

The two aspects make the modeler confident that the refined model is the correct implementation of the abstract model. Next, we will delve into the concept of behavior verification using implementation relations.

$$
\begin{aligned}
&VARIABLES\ openTimeOut\\
&Model \triangleq INSTANCE\ circuitBreakerAbstract\\
&InitImp \triangleq Model!Init \wedge openTimeOut = 3\\
&OpenImp(event) \triangleq Model!Open(event) \wedge UNCHANGED\langle\ openTimeOut\ \rangle\\
&HalfOpenImp(event) \triangleq state = OPENED \wedge event = AttemptReset\\
&\qquad\qquad \wedge openTimeOut = 0 \wedge state' = HALFOPEN\\
&\qquad\qquad \wedge openTimeOut' = 3\\
&TimeOut \triangleq state = OPENED \wedge openTimeOut \geq 0\\
&\qquad\qquad \wedge openTimeOut' = openTimeOut - 1\\
&\qquad\qquad \wedge UNCHANGED\langle\ state\ \rangle\\
&CloseImp(event) \triangleq Model!Close(event) \wedge UNCHANGED\langle\ openTimeOut\ \rangle\\
&FromOpenToHalfImp(event) \triangleq Model!FromOpenToHalf(event)\\
&\qquad\qquad \wedge UNCHANGED\langle\ openTimeOut\rangle\\
&NextImp \triangleq \exists\ e \in Success, Failure, AttemptReset : HalfOpenImp(e)\\
&\qquad\qquad \vee TimeOut \vee OpenImp(e) \vee FromOpenToHalfImp(e) \vee CloseImp(e)\\
&SpecImp \triangleq InitImp \wedge \Box[NextImp]_{\langle\ state, openTimeOut\ \rangle}\\
&THEOREM\ SpecImp \implies Model!Spec
\end{aligned}
\tag{2.3}
$$

Transitions systems are suitable for modeling software systems at various abstraction levels. Bisimulation, simulation, and trace inclusion are among important implementation relations [BK08] that compare two models of one software system, in particular, to check if the transition system Sys is the correct implementation of the transition system $Spec$. Bisimulation and simulation respect the branching-time view, while trace inclusion respects the linear-time behavior view.

Bisimulation aims to determine whether two transition systems, denoted as TS_1 and TS_2, can simulate each other step-wise. Considering the branching-time view, bisimulation establishes relationships between states that mutually simulate each other's next step transitions. In contrast, in a simulation relation, one state must simulate all next step transitions of the other, but the reverse is not necessarily true [BK08].

Trace inclusion relation is weaker in the sense that considers **observed** transition. Trace inclusion has a linear-time behavior view. Figure 2.5 shows the hierarchy among implementation relations. Bisimulation $TS_1 \sim TS_2$ implies simulation relation between the two transition systems $TS_1 \preceq TS_2$. Furthermore, if $TS_1 \preceq TS_2$ and TS_1 has no terminal states then $TS_1 \subseteq TS_2$. The following provides a detailed explanation of simulation and trace inclusion relations.

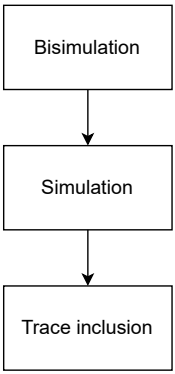

Figure 2.5.: Implementation relations hierarchy based on [BK08]

Consider two transition systems $Spec$ and Sys. Sys is the refined model of $Spec$. In other words, Sys implements $Spec$. Simulation relation is one implementation relation often defined to verify that Sys implements $Spec$ correctly, which means that Sys must simulate all the stepwise behavior of the $Spec$. Simulation relation $\preceq$ is a preorder relation between two transition systems, $Spec$ and Sys. $Spec \preceq Sys$ reads as system simulates specification. Precisely, A simulation for $(Spec, Sys)$ is a binary relation $R \subseteq S_{spec} \times S_{sys}$ such that

A) $\forall\, s_{spec} \in I_{spec} \,.\, (\exists\, s_{sys} \in I_{sys} \,.\, (s_{spec}, s_{sys}) \in R)$

B) for all $(s_{spec}, s_{sys}) \in R$

1) $L(s_{spec}) = L(s_{sys})$ and

2) if $s'_{spec} \in Next(s_{spec})$ then there exists $s'_{sys} \in Next(s_{sys})$ with $(s'_{spec}, s'_{sys}) \in R$.

The formula **A** means that every initial state in the specification must correspond to an initial state in the implementation. However, the opposite is not necessarily true, as the implementation may contain additional initial states not captured in the design specification. Condition **B** shows that every successor state in the specification has a corresponding successor state in the implementation, and the opposite does not need to hold.

It is often the case that implementation model Sys has more added details and concrete values than abstract model $Spec$. Hence, to show that formula $L(s_{spec}) = L(s_{sys})$ holds, there needs to be a sequence of refinement mapping functions [Lam02] that maps state values of Sys to state values of $Spec$. Let variables for states S_{spec} to be $V = \{v_1, v_2, ..., v_n\}$ and variables for states S_{sys} to be $C = \{c_1, c_2, ..., c_m\}$. The sequence of refinement mapping functions is defined as substitution $v_i \leftarrow F_i(c_j)$ such that $i \leq n$ and $j \in C$. The function domain F is one or more declared variables in C. Second, the next step in Sys execution changes the mapped state exactly as the next step in $Spec$. Figure 2.6 illustrates *refinement mapping* and *step simulation*.

Trace inclusion $\preceq_{tr}$ is another implementation relation weaker than the simulation relation that often has been studied in the model-based testing

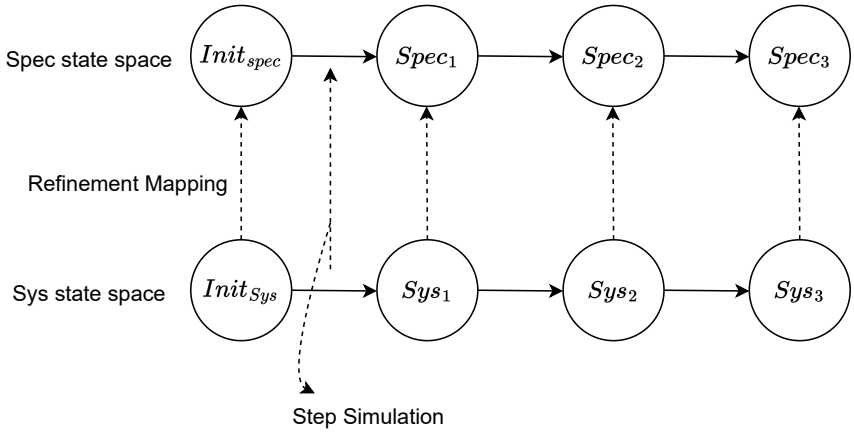

Figure 2.6.: Sys correctly implements $Spec$

community. Weaker in the sense that trace inclusion focuses on specific observed behavior rather than overall behavior. The idea is to test the software system against its high-level specification model and observe only traces (i.e., runtime traces) that are a subset of all traces from model specification, $traces(Spec) \subseteq traces(Sys)$, means that the system correctly implements its specification. Often, checking whether or not all traces of source code conform to the system specification is infeasible for large code bases. Hence, approaches that check individual traces of the system at runtime scale better with the disadvantage that the approach is not exhaustive, such as the model-checking verification technique.

To extract traces from the running system, one must add extra code to the source or binary. This process of adding extra code to an existing system and analyzing an observed trace connects this dissertation to Reverse Engineering (RE) research.

2.2. Architecture Reconstruction

Analysis of an existing software artifact, such as source or binary code, is a core activity in reverse engineering process [CPC11][1]. The analysis results will be used to extract different views suitable for software engineers or tools. For instance, software engineers may use extracted views to understand system behavior. The views are not necessarily at the same level of abstraction compared to implementation. For instance, system observable data typically includes detailed internal or external functional calls. However, using analysis techniques, extracting a sequence of interactions among architecture-level components from observable data is possible. The abstraction level of the views depends on the extraction goal.

Canfora et al., in [CPC11], classify four types of analyzers for software artifacts, including static, dynamic, hybrid, and historical. The notion of software artifacts in the paper refers to system implementation. However, the concept of software artifact in RE is not limited to implementation. Other artifacts are, for example, design documentation and metadata in software repositories. Other artifacts are outside the context of this dissertation, as the focus is on understanding whether third-party implementation behaves as intended. One of the applications of reverse engineering is architecture reconstruction. Before explaining the system implementation's static and dynamic analysis, the following section discusses the definition of architecture and related views.

2.2.1. Architecture and Architecture Views

Bass et al., [BCK12], define the software architecture of a system as "the set of structures needed to reason about the system, which comprises software elements, relations among them, and properties of both." Often, these structures are represented by views that address the concerns of stakeholders involved in the development team. In this regard, Kruchten [Kru95] proposed

[1] Before Canfora et al., Chikofsky mentioned the concept of analyzers in his paper [CI90] on reverse engineering.

a 4+1 view, including Logical, process, development, physical, and scenarios to model a software architecture. The following provides details of each view.

- **Logical.** It is mainly concerned with the system's structure through abstract functions aiming to satisfy the requirements. A typical example in the context of UML is a component diagram that describes system components, their interface, and interactions among them.
- **Process.** It mainly concerns a set of abstractions modeling the system's concurrency, distribution, or fault tolerance aspects. State chart diagrams are examples of modeling system design from a process view perspective using UML.
- **Development.** It focuses on the organization of the software modules and packages. An example is a package diagram in UML
- **Physical.** The view considers the association of software elements identified in logical, process, or development views to the physical machine. The view concerns the execution of the software systems. Deployment diagrams in UML describe software architecture from a physical view.
- **Scenarios.** They typically describe a domain problem and show how different elements described in different views work together to satisfy the scenario. Scenarios can be in the form of text or graphical.

From the author of this dissertation's point of view, formal models are architectural models. The main reason is that architecture models are at the level of abstraction that allows reasoning about the quality requirements, which is similar to the aim of the formal methods that concentrate on the abstract level that allows analysis of quality requirements. However, the abstraction level in the formal method community is often referred to as design rather than architecture. Another benefit of considering formal models at the architecture level is that conformance checking is an established field in the architecture evaluation of implemented software systems. Therefore,

investigating the literature on architecture conformance checking provides a broader investigation of existing processes and methods.

Formal method experts often consider all the views when modeling software systems at various abstraction levels for verification purposes. For instance, they must consider main functionality, interactions, concurrency, physical machines, and quality requirements. A few standard graphical notations exist for visualizing transition systems, most likely suitable for verification and human understandability. They are state, control flow, and call graphs. State graphs [BK08], a directed graph, is a suitable representation of transition systems for checking correctness properties. A graph is a suitable representation because it provides a rich set of algorithms for exploring paths, which is needed for verification. Consider $TS = (S, Act, \rightarrow, I, AP, L)$ to be a transition system. A state graph of TS, defined as $G(TS) = (V, E)$. Vertices in $G(TS)$ are states S of the transition system TS and edges are $\{(s, s') \in S \times S \mid s' \in Next(s)\}$.

Control flow and call graphs are other representations for verification and debugging. A CFG is a directed graph where each node shows a basic block of the code, and the edges among the nodes represent the control flow between the basic blocks. Another helpful representation is a Call Graph (CG), a directed graph where each node shows a method, and each edge represents that a source method calls or returns from the target method. Constructing a precise CFG or CG depends on properties that must be checked. For example, it is possible to represent a few assignment statements as one basic block. Hence, the ordering of the statement execution becomes insignificant. However, if the analysis depends on the returned value, the ordering becomes important, and therefore, separate basic blocks are needed for each different assignment statement.

2.2.2. Static and Dynamic Analysis

Comparing two behavioral models representing a system at design and implementation level abstractions is central to BCC. In our work, the models are obtained by the *execution* of design specification and written program. Related to RE practice, at the Implementation level, models are typically extracted using static or dynamic analysis that involves program execution. Hence, the rest of the section aims to provide a unified understanding of *execution* meaning. At the design level, the notion of model execution in the context of transition systems refers to the execution of actions and the assignment of values to variables. Model-checking tools provide a so-called reachability graph that describes all reachable states starting from the initial state.

Static and dynamic analysis are two key RE activities that involve program execution. Examples of static analysis are type checking by compilers, identifying unintended behaviors, or code visualization for developer understanding. On the other hand, dynamic analysis involves tools like profilers and execution visualizers. It's important to note that the definition of static and dynamic analysis often incorporates the notion of *execution*, which can be misleading. For instance, at the implementation level, static analysis symbolically executes the source or byte code for analysis purposes. In contrast, dynamic analysis executes the *binary*, including software packages and necessary hardware, with actual values.

Nielson et al. [NNH99], define static analysis as follows: *"compile-time techniques for predicting safe and computable approximations to the set of values or behaviors arising dynamically at run-time when executing a program on a computer."* The application areas the authors consider are mainly compilers and identifying unintended behaviors. The notion of *safe* is in the definition because of the source code transformations that are done by compilers. So, the analysis code (inside the compilers) must be safe. The term *approximation* is used to show the common limitation among many static analysis techniques in the considered application areas for providing exact answers [NNH99]. For example, identifying all dependencies among

statements or data is difficult because function calls to different libraries make dependency computation expensive. Hence, the analysis has to exclude certain aspects, which makes the answer an approximation.

Nethercote [Net04] uses the above definition of static analysis to define dynamic analysis as *"dynamic analysis offers run-time techniques for remembering the approximations to the set of values or behaviors seen so far when executing a program on a computer. "* The notion of approximation is a synonym for abstraction aiming to support different software views. From Nethercote's viewpoint, an important distinction between static and dynamic analysis is that in static analysis, the idea is to *predict* approximations of the program's future. In contrast, dynamic analysis is concerned with *remembering* approximations of the program's past.

The work in this dissertation uses dynamic analysis for extracting behavior models, i.e., models are constructed from observed values at runtime of the system. There are two main reasons. Firstly, considering that third-party components have already been implemented, third-party component designers can use our solution to run existing test cases and observe various system paths for conformance checking. Secondly, application designers can continuously observe the behavior of the third-party component within the system as part of the operation and maintenance process, which requires dynamic analysis. Dynamic analysis often involves instrumentation, which is the subject of the next section.

2.2.3. Instrumentation Techniques

Instrumentation is essential in performing dynamic analysis, which means adding extra code to the source or binary code. The purpose of instrumentation often is to (1) record values that will be needed for constructing different software views or (2) extract software views, which the added code can be referred to as *analysis code* [Net04]. Binary can mean machine code or some intermediate representation such as bytecode. Many programming languages such as Java, Scala, and C# use a virtual machine to translate bytecode to the machine code. On the other hand, source code refers to

programming language constructs such as classes, methods, and control statements.

Instrumentation activity has four main characteristics: automatic, manual, build-time, and load-time. In automatic instrumentation, it is possible to specify *what has to be injected as the extra code*, often in a different language than the target source code. Afterward, generators transform and add the extra code to suitable source or binary code locations. A notable example is the Aspect-oriented programming framework.

In build-time source code instrumentation, all the necessary extra code is added to the source files either manually or automatically. After the build process is completed, the binary includes the original and the instrumentation code. Build-time binary instrumentation happens after the build process is completed, which is often suitable for scenarios where the source files are unavailable. Load-time instrumentation occurs during the program's run. An internal or external process regarding the program is responsible for adding the instrumentation code. Load-time instrumentation is useful for scenarios where some classes are needed for dynamically loaded instrumentation.

2.3. Architecture-based Conformance Checking: Process View

This Section first fixes a conformance definition and then presents the conformance checking process and existing methods for architecture-based conformance checking, including (1) process view, (2) logical view, and (3) quality requirements.

2.3.1. Conformance

In the context of formal methods and software testing, there are two terms called static and dynamic conformance [CFP96]. Given a software system and its specification at a higher level of abstraction, *dynamic conformance* is defined as implementation relations such as the simulation relation. When a specification is parameterized, *static conformance* refers to correctly instantiating a specification such that values of the parameters are allowed a

combination of implementation options. To avoid overloading the term *static and dynamic*, this dissertation does not use the notion of *dynamic conformance* but architecture-based conformance from the process view perspective.

The term conformance has been used in the context of software architecture evaluation. Considering what constitutes a software architecture, meaning of conformance defined by de Silva et al., [SB13] as "an implementation that satisfies the constraints specified in its prescribed architecture." Moreover, the authors mention that satisfaction of constraint can relate to (1) static structure, which can be represented by, e.g., logical view, (2) dynamic structure, which can be represented by, e.g., process view, or (3) quality of service requirements. Although, it has to be mentioned that the authors did not attempt to provide a precise definition of conformance. The precise conformance definition in the software architecture context likely depends on the different modeling used in each view. In the case of the process view and use of the transition system as modeling formalism, there is a precise definition of conformance through implementation relations as discussed in Section 2.1.5. Regarding quality requirements, the definition of conformance depends on verification techniques such as model-checking or software testing.

Two possible categories of methods for ensuring architecture conformance are (1) *conformance by construction* and (2) *conformance by extraction* [BAE+15]. To go from requirements to implementation, conformance by construction methods incorporate (1) model-driven methodology, such as generative programming, or (2) formal method, such as refinement techniques. In contrast, engineers use conformance by extraction methods after implementing the software system. The extraction is done by employing static or dynamic analysis to create architectural views; for example, source code and runtime traces are used to extract as-implemented architecture views. This dissertation considers conformance by extraction methods and processes because third-party components used by application designers have already been implemented. From now on, the intention is conformance by extraction whenever conformance checking is used.

2.3.2. Conformance Checking Process

Conformance checking methods run based on a common process, which is visible in Figure 2.7. The process defined by Tvedt et al. aims to check whether or not as-designed architecture conforms to its as-implemented architecture [TCL04]. The authors mention that the process utilizes a static view of the system where components and their interactions via interfaces model as-designed architecture. However, one view is likely chosen to describe each step, and the process is most likely similar to other views. Although details of the step are likely to change. For example, the comparison step for quality requirements/metrics concerns whether the as-implemented architecture possesses certain quality requirements. In contrast, the comparison for the static view should find deviations such as missing interaction between two components.

The process suggests collaborating between two roles: the designers of the software system that must be checked for conformity and a separate analysis team. The rationale is that the process should not interrupt the daily activity of the designers' team. Furthermore, the process does not assume that the architectural specifications already exist; in that case, designers need to provide the architectural specifications.

The process starts with specifying the as-designed architecture. The output of the step is specified architecture and the quality requirements (i.e., metrics). This step is optional when an architecture already exists. Extracting the as-implemented architecture is another step that requires implementation artifacts, including source code. After having both as-implemented and as-designed architectural artifacts, the next step is to compare them. The comparison shows deviations, including missing or extra components, and the same for the interactions. The output of the comparison step is a set of conformance violations, which need to be verified by one or two development team members of the system. The reason for the verification step is threefold. First, to ensure that the violations are not identified incorrectly. Second, the step gives feedback to the development team on how well as-implemented architecture matches as-designed architecture. Lastly, the analysis team will

learn how and why violations occurred.

After the violation verification step, the analysis team performs change suggestion steps to remove violations. In some cases, changes are related to as-designed architecture, and others are related to changes in as-implemented architecture, e.g., in the source code. The analysis team is not responsible for applying the changes but for giving feedback to the development team. According to the process, the development team decides what changes must be applied and whether changes are applied to as-implemented or as-designed architecture. After fixes are applied, the steps of architecture extraction and comparisons are repeated to ensure the new changes do not introduce new violations and the previous violations are fixed.

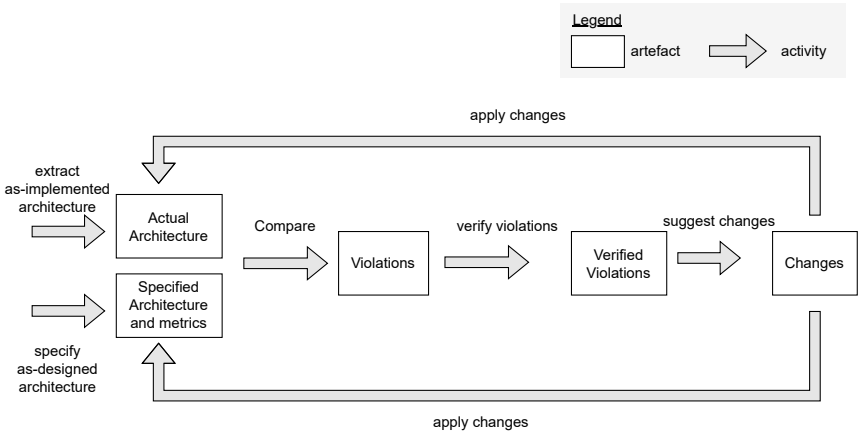

Figure 2.7.: Conformance checking process (based on [TCL04])

2.3.3. Conformance Checking Methods

In Figure 2.7, as-designed architecture is a set of views modeling different aspects of the system and guiding what view is relevant to be extracted from the implementation for comparison. Logical and process views and metrics are likely among the most important for conformance checking. Next is the

explanation of existing conformance checking under the views and metrics. As this dissertation focuses on conformance checking of behavioral models using dynamic analysis techniques, we only briefly explain existing methods for metrics conformance checking and models concerned with the logical view of software systems.

Moreover, this section does not detail program verification techniques such that the input is source code, and the algorithm checks whether or not a program satisfies certain properties, such as deadlock or termination. Such algorithms have been explained under program verification [CHV18]. The general idea is to examine all possible paths that program execution takes and check if the specified properties are violated. Typically, program verification uses model-checking to explore all the reachable states. However, there are techniques such as *symbolic execution* that allow more efficient exploration. Instead of using specific input values, *symbolic execution* operates on symbolic representations of variables, which are expressions that represent all possible values that a variable can take.

It has to be mentioned that there are conformance checking methods that use static analysis techniques. Program verification is an example of such a method.

Metrics/Quality requirements Runtime Verification(RV) concerns checking whether the system's observable behavior at runtime satisfies certain properties. Observable behavior refers to runtime traces from the system. RV is lightweight compared to exhaustive model checking techniques because the check is only in observed behavior. Hence, RV does not guarantee that all possible behaviors of the system satisfy the (un)desired property. LTL is one of the languages that has been used for property specification. LTL has a qualitative notion of time, which means ordering the events/states but no explicit relation to points in time. Hence, other variants of LTL, such as metric temporal logic [Koy90], allow a quantitative notion of time in the specification of properties.

Design by contract (DbC) [Mey97] and *Aspect oriented programming* are runtime verification techniques and likely most used in industry for software development where the specification of property is very close to programming

languages that developers use such as Java. *Design by contract* was first introduced as a built-in feature in the Eiffel language, and later on, extensions have been proposed for Java such as *JML* [BCC+05]. Concerning DbC, the approach is not only limited to runtime analysis, but also compilers of programming languages can perform assertion checking during compile time.

RV methods run based on the process visible in Figure 2.8. The process is defined in [FHR13] where Falcone et al. gave an overview of the RV field, including existing methods. The author's main objective in the paper was not to define and evaluate the process. Hence, descriptions of the roles and guidelines for each activity from a software engineering perspective are missing. However, the high-level phases, including property specification, specification of verification code, instrumentation, execution, and trace analysis, are likely to agree with other surveys in the filed of RV [FKRT21; SSAE19]. The starting phase in RV is the specification of the properties. Although, this step does not have to be done manually and can be automatically derived from source code [ABL02] or runtime traces [EPG+07]. The properties are often system-level and go beyond a single class's scope.

After having a set of useful properties, the following steps are (1) development of property checking algorithm, which in the literature is sometimes called monitor or verification code, and (2) instrumenting the software system. Often, it is possible to generate the verification code from the properties. PathExplorer(PaX) [HR01], Monitoring and Checking (MaC) [KVB+99] and Monitoring oriented programming(MOP) [MJG+12] are exemplary methods that provide generating verification code from the specified properties. The instrumenting step is to produce relevant runtime data for property checking. In the context of MOP, code generation is supported for both instrumentation and verification steps. When both artifacts, including runtime data and the verification code, are at hand, the next step is to check whether runtime behavior satisfies the properties. The output of this step is the verification result and additional information, such as recovery actions in case of property violations. In case trace checking is done along with program execution, it is possible to perform runtime enforcement, such as

self-healing, based on the additional information from the analysis. Otherwise, the system developers perform the corrective actions by introducing a new version.

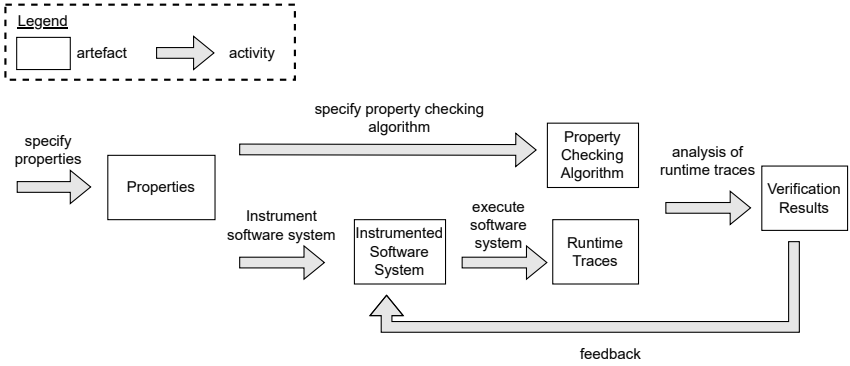

Figure 2.8.: Runtime verification process (based on [FHR13])

Regarding program verification, A class of algorithms is concerned with verifying whether or not a program satisfies certain properties such as deadlock, which has been studied under program verification [CHV18]. The general idea is to explore all possible execution paths and check whether or not the property is satisfied. Typically, program verification uses model-checking to explore all the reachable states. However, there are techniques such as *symbolic execution* that allow more efficient exploration. In symbolic execution, instead of using specific input values, symbolic execution operates on symbolic representations of variables, which are expressions that represent all possible values that a variable can take. On the other hand, *concolic* execution uses actual input to exercise different execution paths, allowing the benefit of the actual execution of the program.

An example of a program verification approach is Java PathFinder (JPF) [HP00], which is specifically designed and implemented to verify Java programs.

Logical View One view considered for architecture-based conformance checking is the logical view, which concerns abstraction units such as class,

module, or components, including their structure, e.g., names, interfaces, and relations among them. For example, a component diagram models the components in the system and how they are related through their interfaces. After extracting the as-implemented component model, *the comparison* shows whether or not relations among components in the as-implemented model deviate from the as-designed component model. The extraction is possible from both source code (static analysis) [PTV+10] and runtime traces (dynamic analysis) [SB13].

Reflexion modeling [MNS01] is the name of the method that Murphy et al. initially proposed for the issue of drifts (sometimes referred to as decay or erosion) among software artifacts at the level of architecture and implementation. The authors describe reflexion modeling for architecture model described by components and their *interactions*. The comparison activity in *Reflexion modeling* is described using *Z* formal notation, and the activity includes three types of relations between as-designed and as-implemented architecture.

Convergence is a relation variable that defines a set of interactions where the as-designed model agrees with mapped interactions from the as-implemented model.

Divergene is a relation variable that defines mapped interactions in the as-implemented model that differ from interactions in the as-designed model, which means that the component designer expected different interactions. and finally

Abscence is a relation variable that defines mapped interactions in the as-designed model that differ from interactions in the as-implemented model, meaning that component designers expected interactions, but they do not exist in the as-implemented model.

Relation Rules [Pos03] is another approach for conformance checking under a logical view that defines allowed or forbidden relations between source and target components. It is possible to combine the definition of such rules with regular expressions. For example, *D* must exist E** means

that a relation must exist from all components starting with the name D to components with names starting with E.

Process View A subset of conformance checking methods propose a solution for comparing the behavior of the two as-implemented and as-designed architectures. Process view concerns aspects such as timing, concurrency, or synchronization among processes in the system.

According to the systematic literature review [MTZ17] by Muram et al., major targeted behavioral models are UML state machines, UML sequence diagrams, and process models, including BPMN, Petri Nets and activity diagrams. Furthermore, the study shows that the majority of the methods (1) rely on state transition for describing the semantics of models and (2) use model-checking tools such as *UPPAAL* or develop a specialized algorithm to perform conformance checking between two models.

Concerning specialized algorithms, In their work [ALC09], Ackermann et al. employ sequence diagrams for modeling as-designed architecture. The authors extract runtime traces from the system and use *an approximate matching* algorithm between two sequences of strings recorded from the trace and the sequence of events from the sequence diagram to compare whether or not the sequence of events (as-implemented) in the trace is a possible sequence of events from sequence diagram (as-designed).

More on the specialized algorithm, according to the state-of-the-art literature review by Dunzer et al., [DSMB20], *trace alignment* "[...] can be considered as the standard [...]" conformance checking technique in the context of the process model. Moreover, the authors explain two inputs to the conformance checking. First is the process model description, e.g., described in Petri Nets and the event log that is assumed to exist already and is relatable to the event sequence described by the model. Trace alignment aims to find an optimal alignment between each trace recorded in the log and the most similar behavior described according to the model. The algorithm identifies deviations between the observed behavior and the intended behavior. For example, γ_1 shows a perfect alignment. Whereas γ_2 shows that

the observed event b does not have the corresponding event at the model level and event e does not exist in the observed behavior.

$$\gamma_1 = \frac{a|b|c|d|e|f}{a|b|c|d|e|f}$$

$$\gamma_2 = \frac{a|-|c|d|e|f}{a|b|c|d|-|f}$$

Concerning model-checking, Arcaini et al., [AGR11] propose *CoMA* (Conformance Monitoring by Abstract State Machine). Abstract State Machine (ASM) [BS03] is the underlying formal method used in CoMA. The core idea is to annotate Java elements, including Class fields and methods, with and without side effects. The annotation aims to link Java implementation to its formal model. Hence, each Java class must have a corresponding ASM model. The observable state is all class fields and pure methods that return a value. Furthermore, a *Java step* is defined as a relation $(s, m, s\prime)$ where s is the starting state of the execution, m is a method of a java class, and $s\prime$ is the last state of the current execution. A *Java step* is a change step when m is an element of changing methods in a Java class. Following is the definition of state and step conformance.

State conformance Observed state s conforms to a state S from ASM specification if the values in the state s conform to the values of state S. While in the paper, the used term is *conform*. However, the meaning is likely to be *equal*.

Step conformance A java change step $(s, m, s\prime)$ conforms with a step $(S, S\prime)$ of ASM if $\text{conf}(s, S) \wedge \text{conf}(s\prime, S\prime)$

In the paper, given an observed computation of a Java instance of class C, O_c, behavioral conformance is defined as *co-simulation*, which is

1. The initial state s_0 of the computation O_c conforms to the initial S_0 ASM computation.
2. every observed change step $(s, m, s\prime)$ conforms with the step $(S, S\prime)$.

Schvimer et al., [SDH20] has explored conformance checking based on the model-checking technique at MongoDB, Inc. The development team uses TLA^+ to specify the consensus algorithm part of the MongoDB program. Later on, the implementation is instrumented to observe the values of the variables corresponding to the specification variables. A separate program creates a trace of the observed behavior fed to the TLC model-checker, which outputs whether or not the observed behavior is part of the intended behavior.

2.4. Discussion of Existing Approaches

This Section lists requirements from the introduced foundation for conformance checking between as-designed and as-implemented architecture models from the process view perspective.

2.4.1. Requirements for Behavioral Conformance Checking of Third-party Components

Behavioral conformance checking combines both disciplines, including RE and MBRAE. The following lists high-level requirements based on the introduced approaches under RV and behavioral conformance checking.

- The method should correctly detect and compare the observed behavior of the third-party code with the specified behavior.

  Dynamic analysis techniques should be supported to extract system traces. Modern software systems are highly dynamic and perform many self-reconfigurations. Hence, dynamic analysis allows the capture of behaviors that can only be determined during runtime. Moreover, support for load-time binary code instrumentation is desirable. At the same time, instrumenting the source code of open-source projects is possible. However, re-building and packaging third-party components requires time and effort. Hence, it is desirable to allow binary instrumentation during load time.

- The method should enhance the understanding of third-party code behavior by providing an intuitive representation of the observed and specified behavior.

  It's crucial to consider supporting end-users of third-party components, as they are not part of the design specification and implementation.

  The method should support both state-based models based on transition system formalism for describing the system behavior.

- The method should automate the conformance checking process as much as possible, reducing the need for manual effort and human intervention. It should provide automated techniques for mapping the observed behavior to the specified behavior, identifying deviations, and assessing the impact of the deviations on system properties.

  Regarding the mapping technique for state variables, the goal is to identify the corresponding state variables between the observed and the specified behavior, which involves analyzing the system's implementation and design specification to understand the meaning and purpose of each state variable. While automating the mapping process can be helpful, the mapping should be explicit and produce artifacts to be used by end-users of third-party components.

- The method should efficiently handle large-scale systems and complex behavior, as lengthy or resource-intensive processes can hinder practical adoption.

2.4.2. Resulting Deficiencies

This Section discusses a list of deficiencies by judging existing approaches against supporting the role of application designers using third-party components, which is listed as a requirement in Section 2.4.1.

Table 2.2.: Overview of selected papers

Research Work	Objectives	Classified as	Deficiencies
JML [BCC+05]	Overview of JML tooling and its applications	RV/DbC	Not designed for specifying system level properties
PaX [HR01]	Combining formal methods with system testing	RV	Requires implementation knowledge, which disallows separation of roles
MOP [MJG+12]	in depth discussion of generating monitors for JavaMop and BusMOP instances of MOP	RV	Requires implementation knowledge, which disallows separation of roles
JPF [HP00]	Translating Java program to Promela for model-checking	Program verification	Requires implementation knowledge, which disallows separation of roles
Reflexion modeling [MNS01]	comparing implementation model to its design	Logical view	Focus of the work is on logical view
Relation Rules [Pos03]	verifying architecture model against architectural rules	Logical view	Focus of the work is on logical view
ARAMIS [Nic18a]	verifying architecture model against architectural rules	Logical view	Implementation knowledge is required

Table 2.3.: Overview of selected papers - Continued

Research Work	Objectives	Classified as	Deficiencies
[ALC09]	comparing implementation model to its design	Process view	Mapping activity is implicit and embedded into comparison
[DSMB20]	Reviewing the literature on process model conformance checking approaches	Process view	-
CoMA [AGR11]	Runtime monitoring of Java programs by linking Java program to its high-level specification	Process view	State variable mapping is implicit and embedded into code
[SDH20]	Model-based trace checking of consensus algorithm in MongoDB	Process view	State variable mapping is implicit and embedded into code

Deviation detection: component level properties RV methods offer checking system properties against system observed behavior. An instance of such a method is *Design by Contract*(DbC), which uses preconditions, postconditions, and invariants to define the expected behavior and contractual obligations of components. DbC emphasizes runtime checks and contract enforcement during program execution to ensure compliance with the specified behavior. DbC does not focus on specifying expected behavior across components and modules, whereas a formal modeling language such as TLA^+ does.

Deviation detection: System level properties Monitoring oriented programming approach [MJG+12] under RV methods allow specification of system-wide properties. MOP allows the specification of properties in mathematical languages such as temporal logic. MOP transforms the specification into verification code and integrates the code into appropriate program points.

In MOP, the specification of properties and expected behavior is at the level of implementation. In the context of third-party components, there is often no specification. To write specifications, a formal method expert requires reading and understanding the source code, which is often infeasible for large code bases.

State Variable Mapping. The state variable mapping refers to establishing a correspondence between the observed behavior models and as-designed models. It involves mapping the state variables of the observed system to the state variables defined in the specification. State variables represent a system's essential characteristics relevant to its behavior. Effective *state variable mapping* is a central activity in behavioral conformance checking that enables a meaningful comparison and identifying deviations between the observed and as-designed behaviors. Creating correspondence between the two models involves understanding (1) the purpose of each state variable

at the design specification level and (2) how the variable is implemented.

Reflexion modeling [MNS01] and *ARAMIS* [NL16] (the PhD thesis [Nic18a]) explicitly support mapping process. In *Reflexion modeling*, the mapping involves (1) analyzing the source code or system execution date, (2) detecting corresponding components, and (3) creating the associations between as-implementation and as-designed components. The output is a set of rules that describes the association and can be stored as an artifact for re-use.

ARAMIS approach [Nic18b] allows interaction rules between components to be defined by software architects, which most likely is the reason why the title of the work contains *behavior-based*. Conformance checking involves monitoring the execution of the system and checking whether or not the interaction rules hold. Moreover, mapping between source code and as-designed architecture is explicit and recorded in XML-like notation.

The mapping process in both studies produces mapping files that are reusable by end-users of third-party components. However, the concentration is not on state-based behavior models.

In behavioral reflexion modeling study [ALC09], extend the reflexion modeling approach to be suitable for architectural models from the process view perspective. The work concentrates on sequence diagrams to demonstrate the creation of behavioral reflexion models. The authors suggest that mapping high-level messages to concrete messages should be automated as the number of messages can go beyond a level that is practical for manual mapping. Hence, they propose their customized and approximate mapping algorithm based on extending an existing string matching algorithm [MM89]. The algorithm serves the purpose of both mapping and checking conformance. The idea core idea is to convert the sequence diagram to a regular expression and find the best alignment between the regular expression and the observed sequence of messages. The mapping algorithm allows for pinpointing conformances and deviations, including omissions and additions.

CoMA [AGR11] uses annotation to map ASM specification constructs to its Java implementation. Whereas Schvimer et al., [SDH20] instrument the source code to generate the observable relevant for the specification model.

Later, a separate script is written to produce TLC-compatible traces to be read and analyzed by the TLC model checker.

This dissertation proposes a configurable state variable mapping model concerning defining concepts and their relationships in state variable mapping between high-level specifications based on transition system formalism and observable behavior.

Third-party Code Behavior Understanding. It is essential to assess how well a behavior conformance checking method enhances the understandability of application designers concerning third-party code behavior. The reason is that end-users of third-party components are not part of the design and development of third-party components.

To judge existing methods concerning understandability, we discuss the outputs of each method and whether or not there have been evaluations for understandability. Studies related to Java MOP [CdR05; CR05; MJG+12] provide no discussion on generating output and how good end-users of the approach interpret the results. However, based on the experience of author of this dissertation, the output should contain

- **Error Message.** When a property violation is detected during runtime, Java MOP **can** generate error messages providing details about the violation, including a trace of calls that led to violations and contextual data such as function arguments and environment variables.
- **Observable data** Java MOP **can** generate log files that record the observed behavior of the program during runtime. These log files can contain information about the execution traces, including method calls, variable values, and property evaluations.
- **Summary** Java MOP **can** generate a summary of results, including monitored properties, evaluation results, and some statistical results regarding efficiency and the number of checked states.

According to [HP00], JPF outputs an error trace after the SPIN model-

checker finishes executing the verification task. The trace shows how the initial state sequence of program statements leads to the error.

The authors of the studies mentioned above did not evaluate how well the results are understandable by end-users of their approach. In particular, understandability becomes important when end-users are not part of the implementation. It is reasonable to assume that works under RV and Program verification target developers with deep knowledge about the program. In contrast, Imune aims to assist third-party component end-users with limited or no knowledge about the component in identifying behavior violations.

Reflexion modeling study [MNS01] provides a few case studies where engineers used the approach. One is a report on the effectiveness of reflexion modeling concerning the understandability of a Microsoft engineer who uses the approach for an experimental re-engineering task of Microsoft Excel. The study setting is relevant to our work as the engineer was not knowledgeable about source code structure, which is often the case for end-users of third-party components concerning program behavior. The study details are reported outside the main paper, here [MN97]. The reflexion modeling paper [MNS01] mentions that "the engineer found reflexion modeling valuable for understanding the structure of the excel and planning for the experimental re-engineering effort." Further details from the case study [MN97] are that the engineer found it valuable to understand how a high-level model is connected to the source code. Based on [MNS01], engineers query the as-implemented model information contributing to convergences, divergences, and absences. The authors continue that tooling comes with both graphical and textual representation. In the Microsoft case study, the engineer found it easier to understand the textual representation.

Relation rule method [Pos03] does not report on the understandability of results generated from the method, including as-implemented module structure and identified violations. However, the authors describe that the tool uses both tabular and graph visualizations for the as-implemented model. In contrast, browsable textual and tabular output is used for violation of rules. The authors mention that use of that the system architects involved in the case study preferred tabular or textual over graphical outputs.

ARAMIS [Nic18a] recognizes the importance of understandability of as-implemented architecture and the deviations. In a case study, the author aims to simulate a situation where a novice architect (a master's student conducting the study) aims to understand previously unknown architecture. However, there is no report on how the applied approach in the industrial case study provided a better understanding of the system behavior for the user of ARAMIS.

In the behavioral reflexion modeling study [ALC09], the authors use two case studies in the context of Systems-of-Systems to evaluate their reflexion modeling approach. In their evaluation results, the discussion is on finding behavioral issues, and there is no discussion on understandability. Concerning visualization, the authors use sequence diagrams to construct the reflexion modeling showing only the deviations, including omission and additions. Two views are provided: (1) a basic reflexion model where the deviation of individual behavior is graphically represented and (2) a high-level reflexion model where deviation concerning the high-level model is shown.

CoMA [AGR11] evaluates the runtime overhead of the approach. It seems essential because conformance is checked online during program execution, not offline. All in all, there is no discussion on the understandability of the approach for end-users. Moreover, the authors do not discuss whether the output is yes/no or if a more detailed explanation is provided. However, as the approach uses model-checking, it is reasonable to assume the capability to produce an error trace showing an example of how violation occurs.

The study by Schvimer et al., [SDH20] at MongoDB is mainly about experience reports from the industry rather than extending existing literature. The study does not report on the visualization and understandability aspects of conformance checking. The development team was mainly interested in whether the implemented behavior of conformance checking aligns with their TLA^+ specification. Hence, catch bugs. However, based on the experience of the author of this dissertation in using TLC model-checking, TLC model-checker version 1.7.1 produces outputs in the form of yes/no with a counter-example when the answer is no.

Chapter 3

Research Design

This chapter explains research design decisions that led to challenge clarification. The research methodology behind this dissertation aligns more with design science as our goal is to develop artifacts, including a BCC method that can be implemented and evaluated to demonstrate their usefulness and effectiveness. For challenge clarification, we performed three research iterations: (1) starting with a literature review to understand definitions and distinctions among meta-qualities resilience, survivability, and elasticity; (2) investigating reasons for unacceptable QoS degradation in modern applications; and (3) using formal methods to specify third-party components' properties and behavior.

The outcome of the first research activity led to the understanding that the execution context within modern applications changes constantly, and meta-qualities emerged to retain system QoS despite changes. A follow-up research activity investigated why modern applications face unacceptable QoS. The observation of author of this dissertation is that modern applications execute many third-party codes, and their assumptions and design decisions are not well documented. The third research activity aims to assist application designers in understanding the behavior of third-party components. As a

solution, we propose design behavior and property specifications of three frequently used architectural patterns, including circuit breaker, retry, and event souring that come with third-party implementation.

We extend the work from the last research iteration to assist application designers in understanding whether or not third-party components behave as intended. After forming research questions, we closely follow design science to identify the challenge and propose a solution. This dissertation proposes the state variable mapping model and explanation techniques for effective conformance checking in terms of correctly identifying deviations and understandability.

We follow software engineering practices and the scientific method. We first devise research questions to guide the literature review and identify existing state-of-the-art approaches concerning the scientific method. Second, we carefully design experiments based on Wohlin's work [WRH+12] to evaluate the claim in this dissertation that is *The Imune method improves behavior understanding and simplifies conducting behavioral conformance checking for different third-party components within the system.*. For understandability, we design a study using humans as subjects, and for correctness, we use a machine experiment. Concerning software engineering, we limit the scope by defining the requirements 2.4.1 and follow agile development by performing a few sprints in designing, implementing, and gathering feedback.

The rest provides more details on each research iteration. Moreover, the end of this chapter presents the research questions that this dissertation addresses.

3.1. Research Decisions and Challenge Clarification

Figure 3.1 illustrated the decision-making structure from research question(s) until research findings. A set of research questions (RQs) is essential for conducting research that effectively addresses a challenge and yields useful outcomes. However, at the beginning of the PhD, RQs and a research gap are often unclear. The following section describes the decision-making process

for each research iteration based on the information presented in Figure 3.1.

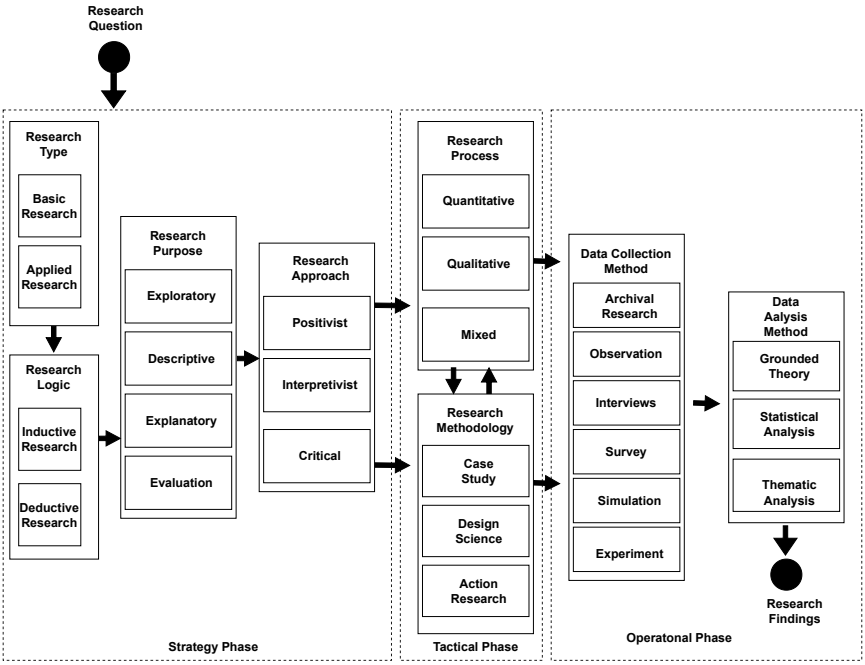

Figure 3.1.: Decision-making structure [WA15].

During the initial stage of clarifying the research question(s) and the gap, we observed that resilience is emerging as a meta-quality attribute in the practices used for developing and operating software systems. This observation was made by discussing with other researchers within the group, reviewing the relevant literature, and keeping track of the development of platforms in the domain of cloud-native applications.

The first decision was to choose basic research from research types, as shown in Figure 3.1, to understand the term resilience better and differentiate it from other meta-quality terms such as elasticity and survivability. The basic research type emphasizes understanding the challenge(s) and

producing knowledge [WA15].

The next step was to select inductive research, as the author of this dissertation did not have an initial theory or hypothesis to test. Inductive research begins with observations, detects theoretical patterns, and forms a general conclusion [WA15]. The research purpose tends to be descriptive [WA15] because the initial research questions were: *what does resilience mean* and *what are the differences to other meta-quality attributes*.

Positivist research emphasizes objective research, meaning that research must be repeatable by other researchers and yield similar conclusion(s) [WA15]. Hence, positivist was the next decision point. The next decision was to pick a qualitative research process, as there was a need to understand other researchers' and practitioners' opinions and perspectives on resilience.

The decision for data collection and analysis was a methodical literature review (archival research) and thematic analysis, presumed to be quicker and give a holistic insight into resilience definition and other meta qualities.

The research findings from the first iteration are:

- Fixing definitions for three meta qualities, resilience, survivability, and elasticity,
- and understanding that all three meta qualities aim to quantify the impact of change within the system execution context.

In the second iteration, the research question has become more focused towards *what are the existing approaches for quantifying resilience, survivability, and elasticity?* and *what are the concepts that characterize change impact quantification*. Decisions for this iteration are similar to the first iteration, and research findings are:

- a Taxonomy to describe existing concepts in the domain of change impact quantification under resilience, elasticity, and survivability,
- and classified existing works based on proposed taxonomy [HKHB20].

The two iterations have produced knowledge in understanding why resilience has emerged, what it means, and what work has been done for

its quantification. Regarding resilience as an emergent quality, the term resilience software systems has been introduced since the 1970s. However, compared to about a decade ago, execution context within software systems *constantly change*, which those changes are planned, such as new deployments, or unplanned, such as outages of the underlying infrastructure. Hence, resilience has emerged to advocate development and operation practices to ensure systems never face unacceptable QoS degradation.

In the third iteration, there was an initial question of *what are possible reasons software systems face unacceptable QoS degradation* despite advancement in both literature and practice for developing and operating resilience software systems and what solution is foreseeable to resolve the issue. Similar to the other iterations, a basic research type fits as the exact challenge was unclear. Choosing the data collection method to fulfill the descriptive research purpose was important. Postmortem documents from industrial cases provide rich information on the causes of system failures, diagnosis, and recovery actions performed by developers and operators. Research findings published in doctoral symposium [Hak20] are summarized as follows:

- Identifying all failure modes during the design and operation of software systems is hard due to the increased usage of service coordination and management offered by third-party services.
- A solution idea is to use a risk-based method to (1) systematically identify faults that lead to failure modes, (2) use scenario description structure for resilience requirement elicitation and characterization, and (3) check if the requirement is satisfied using simulative, model checking or experiment techniques.

The findings from the third iteration led the author of this dissertation to understand that assumptions and key design decisions concerning guaranteed qualities in third-party subsystems are often not documented. Changes within the system execution context may invalidate those assumptions and result in a failure mode. While continuing to collaborate on the solution idea from the doctoral symposium paper under *Cambio project*, author of this dissertation set the objective to *enhance understanding of application*

designers concerning third-party components behavior. To that end, as the first step, the idea was to document third-party components' important quality decisions. The initial research questions are *what are the key design decisions in architectural patterns* and *How do third-party implementations of the patterns violate their guarantees*, which is a more descriptive research purpose than the others. Questions are confined to architectural patterns as application designers often integrate them into the system.

The decision on research methodology is design science, as the aim is to provide artifacts in the form of formal models to capture system design and properties of architectural patterns for verification. For data collection, the decision was to choose archival research (pattern catalogs) to understand the properties and design of the architectural patterns. The decision for evaluation was to choose experimentation on the implementation of third-party subsystems. The main reason was to validate whether or not the findings from model checking hold in actual implementations. The paper [HKF+22] reports the research findings, summarized below.

Findings from [HKF+22]. The main motivation behind the research presented in [HKF+22] is that (1) software systems use many third-party implementations of architectural patterns such as event-sourcing or circuit breaker to cope with changes within system execution context and (2) often important quality decisions are hidden in the codebase and not documented. Therefore, to improve the behavioral understanding of third-party subsystems, the paper proposes a repository of formal specifications of architectural patterns, which contains (1) important quality decisions, (2) a precise description of the pattern's properties, and (3) the intended behavior. The investigated patterns are Circuit Breaker and Event Sourcing.

For specification, the study uses TLA$^+$ as a formal modeling language. The choice of TLA$^+$ was influenced by (1) its adoption in industries such as Amazon [New14] and MongoDB [SDH20], (2) tooling support, and (3) readable specification. All three factors made TLA$^+$ the most suitable choice for our research.

Figure 3.2 illustrates the proposed method for using formal methods in specifying the third-party implementation of architectural patterns, including

(1) pattern selection and review, (2) pattern specification, (3) verification, and (4) pattern specification review. Figure 3.2 shows the four steps that engineers who are experts in formal methods perform for each pattern. The method outcomes are (1) a verifiable model and a (2) description of risk(s) and a decision on sensitivity point(s), which will be stored in the repository of formal models of architectural patterns.

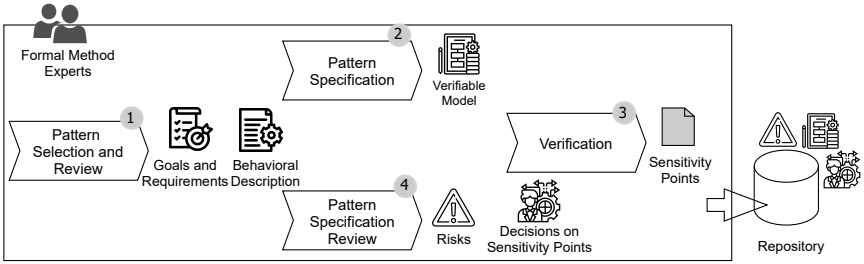

Figure 3.2.: Architectural pattern specification method [HKF+22].

The selection of patterns is based on three criteria: (1) application designers frequently use the pattern in the cloud-native domain, (2) the properties and pattern's design must satisfy are not readily evident from the pattern description or its implementation, and (3) the pattern should have different implementations to compare their important quality decisions or provide many configuration parameters. Circuit Breaker, Retry, and Event Sourcing are three architectural patterns often used in cloud-native applications. Circuit Breaker comes with different implementations such as *Resilience4j* (for Java programmers), *Polly* (for .Net), and *Hystrix*, which is developed by Netflix but is not being maintained anymore. Retry is supported by *Resilience4j* and *Polly*, and finally, Event Sourcing comes with *Kafka* and *EventStoreDB* implementations, for example. Moreover, each pattern implementation has multiple configuration parameters.

The pattern review phase consists of deriving goals and requirements and a high-level description of pattern design that satisfies derived requirements. The artifacts from the first step are inputs to the pattern specification. There

are a few modeling languages available that are suitable for system specifications.

Results.

- The Circuit Breaker pattern is ineffective in preventing further degradation when many threads are active.
- The event storage design in event sourcing fails to prevent data integrity when all nodes fail, and the newly joined leader does not have the most recent data

In continuing our goal to assist application designers in understanding whether or not third-party components behave as intended, we devised research following design science methodology and incorporating. The following section presents the questions that guide the research presented in this dissertation.

3.2. Research Questions

This thesis addresses the following research questions.

- *Research Question 1.* How to extend existing BCC methods to incorporate the role of application designers for effective conformance checking of third-party components?

  Hypothesis. The development of extensions to address the application designer role simplifies behavioral conformance checking of different third-party components within the software system.

- *Research Question 2.* How does applying the method enhance the understanding of third-party code behavior compared to established practices, such as code inspection?

  Hypothesis. The application of the method improves the understanding of third-party code behavior compared to established methods such as code inspection.

Chapter 4

Imune

This chapter presents **Im**prove **UN**derstanding b**E**havior (Imune) method aiming to address the research question 1, *How to extend existing BCC methods to incorporate the role of application designers for effective conformance checking of third-party components?* The rest of this introductory text summarizes what is explained throughout the chapter.

Imune deals with existing formal design specifications of third-party components and observable data from runtime third-party subsystem execution. Therefore, Imune supports three separate roles, including (1) formal method experts who use mathematically based languages to specify behavior and properties of a third-party component design, (2) third-party component designers who are responsible for developing and maintaining the third-party component, and (3) third-party components end-users who integrate the code into their applications and either report issues such as failures or resolve them by themselves.

Imune provides a configuration model to describe the state variable mapping scenarios. Currently, the modeling language is *YAML*, a human-readable data language. There have been two other options: developing a domain language and choosing other data languages such as *JSON* or *XML*. The

advantage of using *YAML* is that the language is human-readable and provides a relatively more comprehensible representation of mapping rules. On the other hand, exploring a domain language requires close collaboration with the relevant stakeholders, which in this case, are three. We have not explored that direction as there was no evidence to convince us that a domain language is needed. Such evidence relates to the lack of efficiency and effectiveness of users creating state variable mapping models using an existing data language.

The mapping model supports the specification of state functions described in the refinement mapping Section 2.1.5 of Foundation. The existence of formal specification and runtime traces are the two preconditions for creating mapping models. State variable mapping involves creating the correspondence between variables at the specification level and variables at the implementation level. TLA$^+$ is an expressive specification language. Hence, to cover the most common mapping scenarios, one should explore different specifications to familiarize oneself with the types of variables that are often used.

Exploring five specifications, including Circuit Breaker, authored by the dissertation's author, as well as Producer-Consumer, ReadWrite lock, AsyncIO, VCHAN protocol, and Voldemort key-value data storage, each developed by different individuals, sheds light on how people choose variable types. The mapping model, employed by Imune, identifies computed variable values through TLC model-checking, encompassing Set, Sequence, Integer, String, and function types.

Examining example specifications reveals that design specifications may incorporate auxiliary variables like history variables. These variables capture past values, aiding in property verification and decision-making. For instance, in the Circuit Breaker scenario, recording the number of calls is crucial for enabling the Circuit Breaker to transition to the 'Open' state. Accurately characterizing variable values is paramount for effective mapping, yet it's only one aspect of the state variable mapping process.

Another aspect involves defining mapping functions that substitute specification variable values with corresponding observable ones. Imune introduces

two concepts — Spec Variable Characterization and State Mapping Function — to address both aspects.

Additionally, to state variable mapping, an algorithm is required to detect and visualize deviations. The algorithm steps follow simulation relation with the modification described in Section 4.2. The idea is to check whether or not an observed step-wise behavior from system execution mimics a step-wise behavior from specification and reports on states that are *conforming*, *missing*, *diverged* and *new*. The following are details on the process model, a description of the roles and artifacts, and an instantiation of the process model for the Circuit Breaker example.

4.1. Imune Development Process

This section details Imune-supported roles, their activities, and involved artifacts.

Overview. Imune distinguishes three roles: formal method experts, third-party component designers, and the end-users of third-party components. Figure 4.1 illustrates all the roles and the involved artifacts.

Three main artifacts are part of the Imune instance for outputting information to assist end-users of third-party components in understanding specification behavior and runtime trace from the system execution. The first artifact is the *Reachability Graph* generated by model-checking. The *Reachability Graph* is the entire reachable states specified according to the design specifications of the third-party components. Creating the formal specification requires formal method experts knowledgeable about mathematical modeling languages and verification techniques. The second is *Runtime System Observable*, which describes observed data from the runtime of the third-party subsystem execution. Observable data must cover the specification variables.

Therefore, it is required that the third-party source code or binary be

instrumented to emit relevant data for trace inclusion checks. Ideally, to identify the instrumentation points, a collaboration between the designer of the formal specification and the designers of the third-party component is required. Identifying instrumentation points is currently a manual activity. The third artifact is the state variable mapping model, which relates the third-party subsystem observable to specification variables. The role of creating the artifact is that of a formal method expert, as knowledge about the meaning of the variable values and proper mapping rules is required.

The Imune instance uses the input artifacts to output (1) trace conformance graph and (2) specification variables description, including domain values of each variable and step-wise changes in values of each variable.

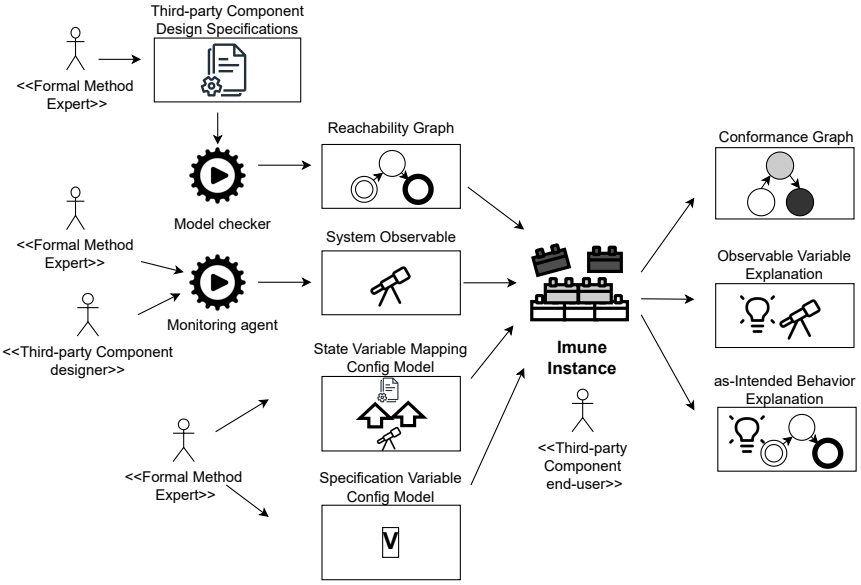

Figure 4.1.: Roles and artifacts in Imune

Process Model. The Imune process is intended to support operation and maintenance process models among system lifecycle processes. According to the ISO/IEC/IEEE 15288 [ISO15], the purpose of the operation process is to

sustain system services by identifying and analyzing operational anomalies related to agreements and stakeholder requirements. The maintenance process runs in parallel with the operational process, and its purpose, according to ISO/IEC/IEEE 15288, is similar to sustaining the system's capability to provide a service. The two essential activities under operation and maintenance processes are recording evidence that shows system service(s) does not comply with the requirements and reviewing them for corrective actions, respectively. Imune aims to fit into this process.

Figure 4.2 illustrates the Imune process model, including the roles, activities, and flow of artifacts among activities. The dashed line separates the development of Imune artifacts (left side) and the use of Imune by end-users of third-party components (right side). On the left are activities, including formal design specification of the third-party components, Code review, and Instrumentation planning mapping specification.

Formal method experts specify the requirements and design of the third-party components. The expected modeling formalism is a transition system. The role is to ensure the design satisfies the requirements by performing verification. There are two essential outputs from this activity. The first is the reachability graph produced by model-checking tools such as TLC. The second is state variables, which are input to the Code review activity.

Code review aims to examine how state variables are implemented and to instrument the code to emit relevant observables during runtime that reflect the values of specification variables.

Ideally, both formal method experts and third-party component developers collaborate. If the source code is inaccessible, reviewing the existing observables during subsystem execution can be an alternative. Quality third-party components often provide extensive observable data. It has to be noted that the coverage of behavior depends on observable data.

Collaboration between experts and developers ensures comprehensive coverage. Formal method experts understand variable meanings, while third-party components assist in locating variable implementations in the code.

The instrumentation step's outcome is (1) a monitoring agent (a separate

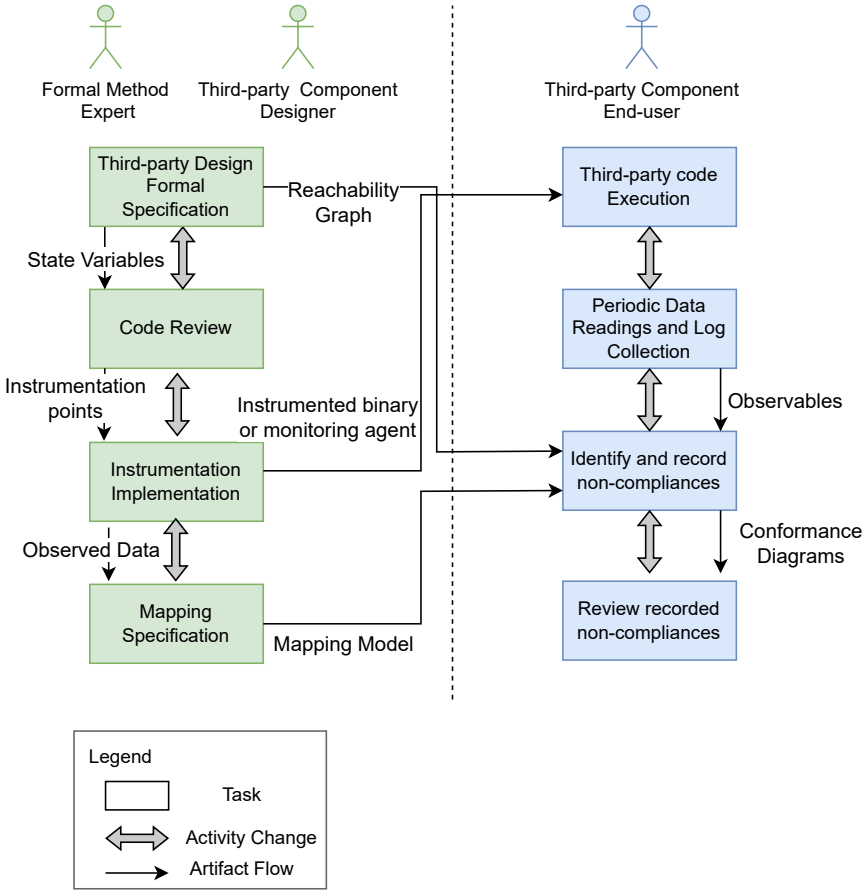

Figure 4.2.: Process Model of Imune, integrated into operation and maintenance activities of applications composed of third-party components

program or pre-compiled binary with additional code) and (2) observable data, essential inputs for the mapping specification process.

Mapping specification involves establishing relations between observable data and each state variable. This task is ideally performed by a formal

method expert familiar with each state variable's meaning. End-users of third-party components rely on three key artifacts: (1) a mapping model, (2) an instrumented component or a monitoring agent, and (3) a reachability graph. The process of state variable mapping encompasses three main activities: code reviewing, instrumentation, and mapping specification.

Third-party code execution, conducted by application designers, involves collecting observable data, typically in testing or production environments. This data serves as input for identifying and recording non-compliances, a fully automated task. Essential resources for this activity include the mapping model, reachability graph, and observable data. The output is a conformance graph depicting conforming, diverged, missing, and new states.

Reviewing non-compliances entails analyzing the results generated by Imune regarding discrepancies between as-implemented and as-intended behaviors and subsequently making corrective or optimization decisions. This process requires a thorough understanding to ensure informed decision-making, often necessitating communication with the development team of third-party components. Ultimately, the main outputs of this activity are a comprehensive understanding of the system's behavior and informed decision-making.

Imune Process Model Applied to Circuit Breaker Example. This section describes the Imune process model based on the Circuit Breaker example. A short history is given concerning the formal design specification of third parties that implement Circuit Breaker. Circuit Breaker design specification is written by the author of this dissertation [HKF+22]. The specification is not written with conformance checking in mind, which means modeled variables and behavior are based on pattern description and not a particular implementation. For example, the Resilience4j library considers three *closed*, *open* and *half-open* states, which is the case in most of the pattern descriptions in the literature such as [Nyg18]. Whereas Hystrix only has two states of *open* and *close*. Both libraries are implementations of Circuit Breaker for Java programs. Next, we explain the process of state variable mapping for two

variables *circuitBreaker* and *calledInState*[1]. The two variables are important for verifying Circuit Breaker requirements, which is why they have been selected. The following describes the variable purpose and possible values that each takes.

- **circuitBreaker.** Models three possible states in the Circuit Breaker, including *closed*, *open* and *half-open*. While initially, the possible values for the variable were either 0, 1, and 2, for clarity, it is changed to string values closed, open, and half. This change is part of a good visualization for the correct interpretation of the states by end-users.
- **calledInState.** The variable records the current state of the *circuit-Breaker* when the call to target service is returned. For example, the sequence $\langle closed, closed, opened \rangle$ showed two calls against the target service when the state of the Circuit Breaker was closed and one when the state was open.

The next step is to find the corresponding implementation of the two variables. We picked the Resilience4j implementation mainly because the implementation is the successor of Hystrix and is under active development. By default, the implementation provides a set of metrics, including state and a few aggregated metrics, such as the total number of successful, failed, or ignored calls. The current observable was not enough for getting values of the variable *calledInState*. The decision is to instrument all the methods and emit the returned values by the method on exit. We could review the code and find suitable program locations for the instrumentation. Expected effort for reviewing the large codebase and requiring recompilation of the library after instrumentation, instead the author of this dissertation decided to instrument all the methods concerning *io.github.resilience4j.circuitbreaker.* package from the binary at load time using standalone monitoring agent.

Listing 4.1 shows the implementation of the *Agent* class. The *premain* method is the entry point of the Java Agent, which is responsible for installing

[1]The name of the variable in the paper is *executed*, which is now changed to *calledInState*. Mainly because the meaning is more apparent than the former name.

the agent on the *JVM*. The class configures an *AgentBuilder* to transform all classes by applying the CircuitBreakerInspector advice to each method.

```
public class Agent {
  public static void premain(String agentArgs,
      Instrumentation inst) {
    String packageMatcher = "io.github.resilience4j
        .circuitbreaker.";
    AgentBuilder.Default agentBuilder =  new
        AgentBuilder.Default();

     /** Add a transformer to capture fields,
         methods, and constructors of matched
         classes */
    agentBuilder.type(ElementMatchers.
        nameStartsWith(packageMatcher))
                .transform((new AgentBuilder.
                    Transformer.ForAdvice()
                          .advice(ElementMatchers.
                              isMethod(), Inspector.
                              class.getName())

    /** Apply the agent builder to the
        instrumentation  */
    agentBuilder.installOn(inst);

  }
}
```

Listing 4.1: Agent class that instrument all methods in circuitbreaker package of Resilience4j

Listing 4.2 depicts the prototype of the visitor class *CircuitBreakerInspector*, which instruments all methods upon exit to capture returned values, method names, and class names. Notably, the returned value is *null* for void methods.

```
class CircuitBreakerInspector {
```

```
    @Advice.OnMethodExit(onThrowable = Throwable.
        class)
    static void onExit(@Advice.Origin Method
        methodName,
    @Advice.Return(typing = Typing.DYNAMIC) Object
        result, @Advice.Thrown Throwable throwable) {
      if (throwable == null) {
        String className = methodName.
            getDeclaringClass().getSimpleName();
        System.out.println("Timestamp: " + java.sql.
            Timestamp.from(Instant.now()) + "
            ClassName: " + className +
        " MethodName: " + methodName.getName() + "
            Return value: " + result);
      }

    }
  }
```

Listing 4.2: CircuitBreakerInspector class

We use a demo application showcasing the Circuit Breaker functionality provided by the development team of Resilience4j. We create a test case mocking success and failure requests to trigger Circuit Breaker to switch between all possible states.

The following is the excerpt of the observed data after running the program relevant for the variable *circuitBreaker*.

```
Class: ClosedState,Method: getState,Return value: CLOSED
...

Class: OpenState Method: getState,Return value: OPEN
...

Class: HalfOpenState,Method: getState,Return value: HALF_OPEN
```

The following is the excerpt of the observed data after running the program relevant for the variable *calledInState*.

```
Class: CircuitBreakerOnSuccessEvent,Method: toString,
Return value: ... successful ...
...

Class: CircuitBreakerOnErrorEvent Method: toString,
Return value: ... error ...
...
```

According to the observable, the variable *calledInState* is not directly implemented. The variable is, in fact, an Auxiliary history variable. To state values for the variables need to be reconstructed based on the values of the *circuitBreaker* variable whenever a call to target service is recorded, which is exactly the meaning of the variable *calledInState*.

The state mapping functions for the variable *circuitBreaker* is:

$$circuitBreaker \leftarrow F_{circuitBreaker}(circuitBreakerImp)$$

The *circuitBreakerImp* variable represents implementation of the *circuitBreaker* variable through return values from methods *getState* of classes *ClosedState*, *OpenState* and *HalfOpenState*.

The function $F_{circuitBreaker}$ substitutes values of the variable *circuitBreakerImp* observed at runtime with values of the variable *circuitBreaker* at the specification level. Listing 4.3 is a pseudo-Java implementation of such a mapping function.

```
if (circuitBreakerImp = CLOSED) {
  return closed
} else if (circuitBreakerImp = OPEN){
  return opened
} else if (circuitBreakerImp = HALF_OPEN){
  return half
```

```
} else {
  return circuitBreakerImp
}
```

Listing 4.3: circuitBreaker mapping function

The state mapping function for the variable *calledInState* is:

$$calledInState \leftarrow F_{calledInState}(circuitBreakerImp, calledInState)$$

The *calledInStateImp* variable represents implementation of the *calledInState* variable through returned values from *toString* of classes *CircuitBreakerOnSuccessEvent*, and*CircuitBreakerOnErrorEvent*. Listing 4.4 is a pseudo-Java implementation of such a mapping function.

```
if (calledInStateImp = successful) {
  return circuitBreakerImp
} else if (calledInStateImp = error){
  return circuitBreakerImp
} else {
  return calledInStateImp
}
```

Listing 4.4: calledInState mapping function

Figure 4.3 shows part of the trace inclusion checking visualized as a conformance graph. The initial observed state from system execution at runtime shows that the variable *circuitBreaker* is *closed* and there has been one call, which is done when the state of Circuit Breaker is in *closed* state, modeled by the variable *calledInState*. The next observed state has the variable *circuitBreaker* with value *closed* and the variable *calledInState* with the sequence of values $\langle\, closed, closed \,\rangle$, the state is represented as diverged meaning that it is not possible according to the specification to go directly from the previous conforming state to the current state. Hence, the current state is marked as diverged, and the missing states are visible in the outputs. The explanation is that according to the specification, calling the target

service can not happen immediately and requires recording previous calls and then obtaining *permission* for making a new call.

After presenting the Imune development process and its integration into the system operation and maintenance lifecycle process, this section introduces Imune's meta-model. Imune's core meta-classes are an abstract model for implementation relations, an abstract model to describe traces from third-party components, a model for refinement mapping functions, and a model for characterizing variables from formal specifications.

Imune considers that an application is composed of many third-party components, and the formal design specification and implementation of the component are possibly done in different organizations at different geographical locations. In order to support behavior understanding of multiple third-party components, the formal method expert role provides two models in addition to the formal specification. They are variable characterization and refinement mapping functions models, which the following subsections explain.

4.1.1. Spec Variables Characterization

In Imune, the formal method expert role who creates the design specification of third-party components uses the *data language, yaml* to serialize metadata about variables in the specification, including (1) regular expression for identifying values and extracting the variable values from the generated reachability graph, (2) boolean values to serialize whether or not values of a variable is of type set or tuple, and finally (3) name of the variables.

For Imune, the type of values assigned to variables are necessary because of (1) correct output formatting. For example, the interpretation of a set of values and a tuple of values are different, and (2) equality check between observed values at runtime and computed values of the formal design specification by the model checker. The following explains the value types that Imune considers.

The foundation section introduced the notion of a state in transition system formalism, i.e., assignment of values to variables. For Imune to correctly

format, it is required to understand the types of values that a model checker can compute from a specification written in temporal languages. As the primary focus of this dissertation is on TLA+ language and TLC model checker, the following introduces the value types that TLC computes based on [Lam02], i.e., called TLC values, including

1. Boolean, including *TRUE* and *FALSE*
2. Integer such as 3 and -1
3. String, e.g., "$closed$"
4. Model values that are introduced in *CONSTANT* statement such as $a1$, or $p1$. Model values with different names are assumed to be different.
5. Finite set of comparable TLC values, e.g., $\{2, 1\}$
6. a function whose domain is TLC value such that $f[x]$ is a TLC value for all x in domain f.

Two TLC values are comparable if and only if the TLA$^+$ semantic determines that they are *equal* or not. For example, the set $\{$"$closed$", $24\}$ are not comparable as TLA+ semantic does not determine whether " closed " equals 24. Hence, $\{$"$closed$", $24\}$ is not a TLC value. On the other hand, the set $\{$"$closed$"$\}$ and the set $\{2, 4\}$ are comparable as the number of elements is not equal. Therefore, the two elements set $\{\{$"$closed$"$\}, \{2, 4\}\}$ is a TLC value. According to the value type described in number 5, a tuple that is a finite ordered list of elements is also a TLC value. For example, the tuple $(10, 2, 3)$ or $(TRUE, FALSE, TRUE, TRUE)$ are tuples and TLC values.

Considering the Circuit Breaker example, two of the variables are (1) *circuitBreaker* that takes either "$closed$", "$opened$", or "$half$" String value, and (2) *calledInState* that contain n-tuple values, e.g., in one state empty $\langle\ \rangle$ and another state $\langle$"$closed$"$\rangle$. Figure 4.4 illustrates three states of computed values by the TLC model checker. Imune supports the interpretation of set, tuple and function value types.

4.1.2. Refinement Mapping Functions

To extract from system runtime execution a comparable model with the system design specification, the formal method expert role specifies refinement mapping functions. The mapping model is specified using *yaml data language* notation and serializes two essential meta-data among others, including (1) mapping function, and (2) whether or not the variable is an *auxiliary history* variable.

In the circuit Breaker example, the specification variable *circuitBreaker* takes the String value of either "$closed$", "$opened$", or "$half$". The runtime data from the third-party resilience4j execution logs String value of either "$CLOSED$", "$OPEN$", or "$HALF\_OPEN$" by calling the query method *getState*. The following is the log record
Time: xxx, Method: getState, Class: ClosedState, Return: CLOSED

The refinement mapping function exists and must be specified so that the values computed for design specification substitute the values observed at runtime. The following function takes as domain values the values from the implementation-level variable *circuitBreakerImpl*, and the range of the function is values of design-level specification.

$$circuitBreaker \leftarrow f(circuitBreakerImpl)$$

The function f is the following expression.

$$\begin{aligned} &IF\ (circuitBreakerImpl = CLOSED)\ THEN\ closed \\ &ELSE\ IF(circuitBreakerImpl = OPEN)\ THEN\ opened \\ &ELSE\ IF(circuitBreakerImpl = HALF\_OPEN)\ THEN\ half \\ &ELSE\ circuitBreakerImpl \end{aligned} \tag{4.1}$$

Sometimes, it is impossible to reconstruct the variable values from a record in the log. For example, concerning the Circuit Breaker example. The variable

`calledInState` takes values of type tuple such as

$$\langle \text{“}closed\text{”}, \text{“}closed\text{”}, \text{“}closed\text{”}, \text{“}closed\text{”}, \text{“}opened\text{”} \rangle$$

According to the design specification, after a call succeeds or fails, the current value of the variable *circuitBreaker* is appended to the variable *calledInState*. The value of the variable *calledInState* is a history of the *circuitBreaker* variable value. It is impossible to reconstruct the history from a single record in the observed data at runtime, which shows the system's state at the time of the recorded log. Auxiliary variables are for such cases.

Auxiliary variables, including (1) history, (2) prophecy, and (3) Stuttering, are often required for verifying that lower-level specification is a correct implementation of a higher-level specification [LM17]. The formal method expert role specifies (1) the implementation-level variable as a history and (2) the mapping function using the data language serialization format, which is to reconstruct the values of the variable *calledInState*, Setting the history flag, results in keeping the past values of the variable, meaning that for setting a value for the current state, first previous state value is read and the new value is added, e.g., $\langle \text{“}closed\text{”} \rangle \overset{next}{\rightarrow} \langle \text{“}closed\text{”}, \text{“}closed\text{”} \rangle$

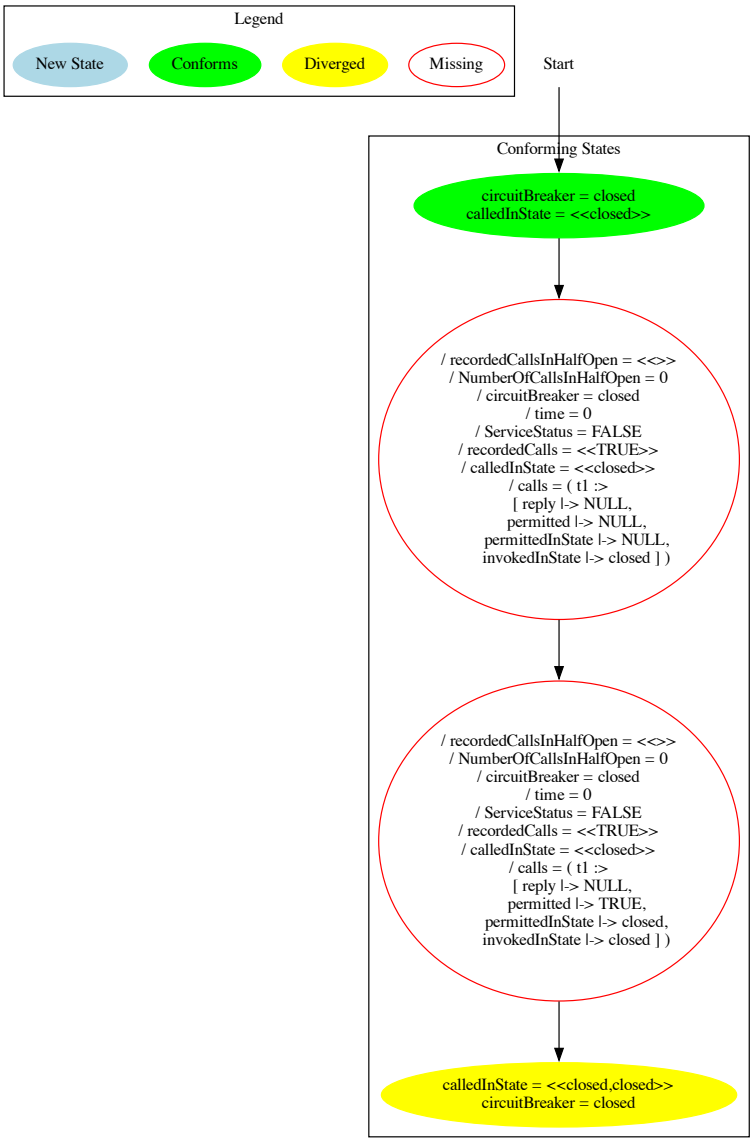

Figure 4.3.: Part of the conformance graph visualization for Circuit Breaker based on Imune tooling.

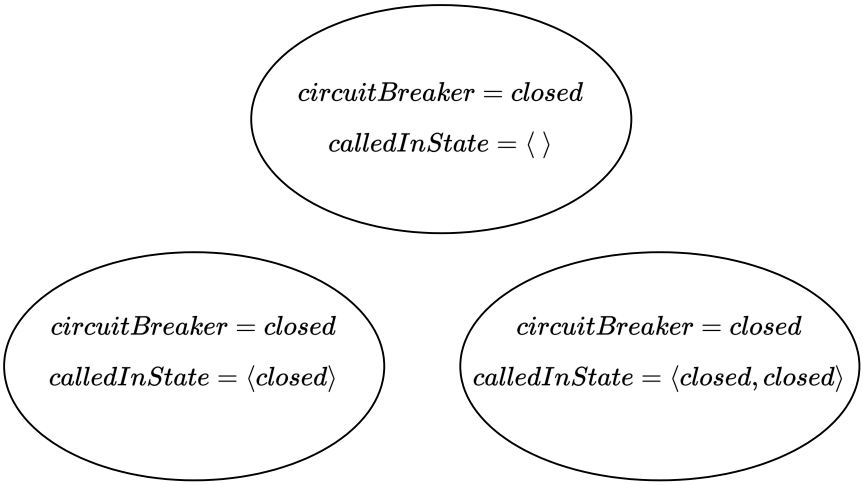

Figure 4.4.: Three sample states computed by TLC, showing assignment of values only to two variables. string quotes are omitted for brevity.

4.2. Implementation Relations

This Section builds upon concepts such as transition systems, paths, and traces explained in Section 2.1. Informally speaking, we want to show that a runtime trace from a third-party code execution is a member of all traces of design specification, i.e., $trace_{Sys}(s) \in Traces_{maximal}(Spec)$. The state s is the first observed state, and the $trace_{Sys}(s)$ is the trace of a finite path fragment that starts in s, e.g., $s, s_1, s_2, ..., s_n$. The state s is not necessarily initial, and the end state s_n is not necessarily terminal.

Often, the execution of software systems does not terminate. The main reason for restricting a trace to be finite is to allow the implementation of a computationally feasible algorithm. $Spec$ in $Traces_{maximal}(Spec)$ is a transition system representing design specifications of third-party components. Maximal refers to the notion of path fragments that can not be prolonged. They are either finite, ending in a terminal state, or infinite. Often, the specification of software systems should be deadlock-free, and hence, path fragments are infinite. More precisely $Traces_{finite}(Spec)$ is defined as

$$\bigcup_{s \in I} Traces_{maximal}(s) \quad and \quad Traces_{maximal}(s) = trace(Paths_{maximal}(s))$$

4.2.1. Static Part

The notion of *transition system*, its representation, and different levels of abstractions are part of Imune core static concepts. In addition, Imune models dynamic features of comparing two transition systems as *operation* within implementation relations.

A transition system includes (1) a definition, (2) a representation of both states and transitions between them, and (3) two different abstraction levels, which are runtime observable and design. The aspects are according to the detailed explanation in the foundation and are illustrated in Figure 4.5. **SpecTransitionSystem** and **ImpTransitionSystem** model design and implementation of third-party subsystems at design and runtime observables levels, respectively. **TransitionSystem** is the base class for both levels, and

the base class references representation class **Graph** and definition class **TransitionSystemFormalism**.

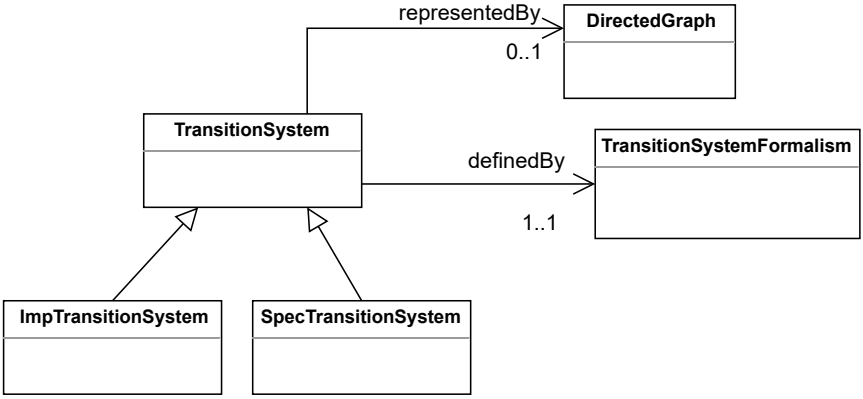

Figure 4.5.: Imune static aspects

4.2.2. Dynamic Part

In addition to its static features, Imune also models the dynamic aspect of transition systems through the **ImplementationRelationOperation**. This operation compares two transition systems of the same third-party components but at different levels of abstraction. Trace inclusion checking, represented by the formula $traces(Sys) \subseteq traces(Spec)$, indicates that *Sys* is the correct implementation of *Spec* (more details in the Foundation 2.1.5).

TraceInclusionOperation models the operation performed within trace inclusion checking, i.e., $\subseteq$. The left side of the formula $traces(Sys) \subseteq traces(Spec)$ is $traces(Sys)$ representing seen traces from the system's runtime. The trace comparison is modeled by association **leftSideTrace**. The right side of the formula $traces(Sys) \subseteq traces(Spec)$ is $traces(Spec)$, representing all traces of the design specification. **TraceInclusionOperation** has a reference to **SpecTransitionSystem** via **rightSideAllTraces** association.

Every implementation relation establishes a binary relation between the

states in the state-space of the *two* transition systems. The scenario is modeled via **ImplementationRelationOperation** base class and the association **binaryStateRelation**. Figure 4.6 shows the explained aspects.

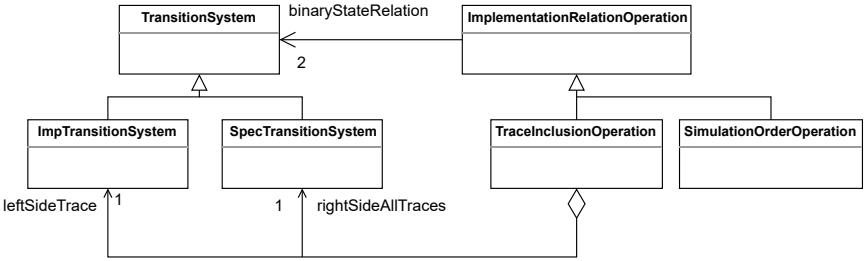

Figure 4.6.: Imune dynamic aspect

The semantic of **TraceInclusionOperation** concept is similar to simulation relation, explained in the Foundation differences that we explain next. The semantic definition of **TraceInclusionOperation** introduces three relations, including $R_{conforms}$, $R_{diverged}$ and R_{New}.

The semantic description of $R_{conforms}$ is the following.

A) init_{sys} . ($\exists\ s_{spec} \in \text{init}_{spec}$. $L(\ \text{init}_{sys}) = L(s_{spec})\ )$ then
$(\text{init}_{sys}, s_{spec}) \in R_{conforms}$

B) for all $(s'_{sys}, s'_{spec})\ \in R_{conforms}$

1) $L(s'_{sys}) = L(s'_{spec})$ and

2) if $s'_{sys} \in\ Next(s_{sys})$ then there exists $s'_{spec} \in Next(s_{spec})$ with $(s_{spec}, s_{sys})\ \in R_{conforms}$

The semantic description of $R_{diverged}$ is the following.

for all $(s'_{sys},\ s_n \in\ S_{spec})\ \in R_{diverged}$

1) $L(s'_{sys}) = L(s_n)$

2) if $s'_{sys} \in\ Next(s_{sys})$ then there exists $s_n \in S_{spec}$ for which a finite walk (e_1, e_2, e_n) exists between vertexes $\{s_1, s_n\},\ i = 1, 2, n$ such that s_1 is the source of e_1 and s_n is the target of e_n with $(s_1, s_{sys})\ \in R_{conforms}$

The semantic description of $R_{diverged}$ is is the following.

for all $(s_{sys}, s_{spec})\ \in R_{New}$

$L(s_{sys}) \neq L(s_{spec})$

4.3. Runtime Trace and Records

Runtime Trace of the system refers to the observed sequence of states during the runtime execution of third-party components. The concept **ImpTransitionSystem** models a runtime trace of third-party components execution. The runtime trace is built by one or more refinement mapping functions that map the variables' values from implementation to corresponding specification variables values. The**StateFunction** represents the concept of the refinement mapping function. The domain of the mapping functions is defined over a set of implementation variables, and the range is defined over the set of specification variables. A mapping function is written as one or more regular expressions modeled as **ExpressionStatement**. Figure 4.7 shows Imune's underlying concepts for extracting a runtime trace by refinement mapping.

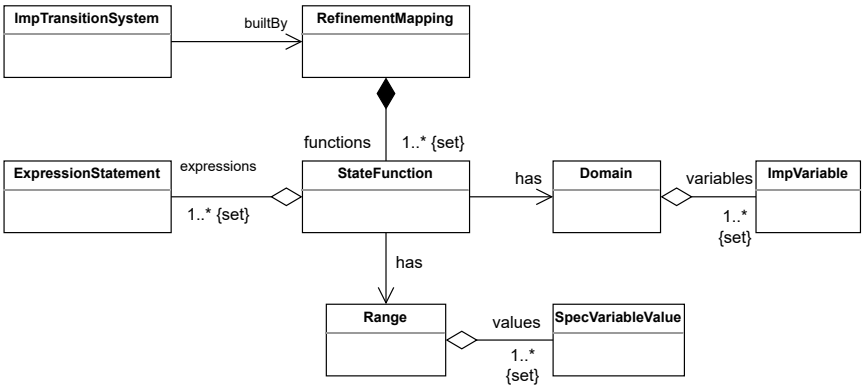

Figure 4.7.: Concepts for Building Implementation Transition System by Refinement Mapping

The observable of the third-party components are recorded in log files. Mapping functions apply to each record to extract a variable value at the observable level with a corresponding specification level variable. Every record has a set of fields, including the name of the executed method and its type in the form of the class name. Return values represent values of the variables at the observable level. The scenario is modeled using **Record** and

Field concepts visible in Figure 4.8.

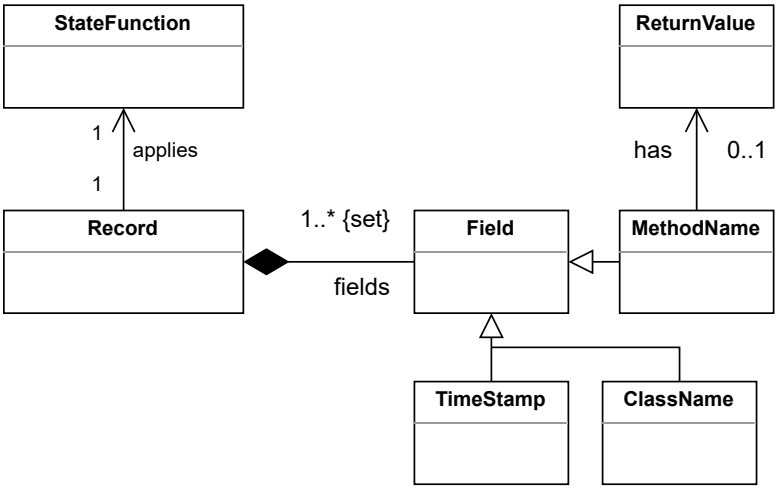

Figure 4.8.: Concepts for Records

4.4. Explaining Specification and Visualizing Deviations

Application designers, end-users of third-party components, are not considered part of the design specification and implementation of the third-party components. Application designers typically consist of professionals with diverse backgrounds, including system software development and operations. While they possess valuable technical skills and domain knowledge, they may not be familiar with formal methods, including modeling by mathematical languages, verification by model-checking, and implementation relations. In such cases, explaining formal design specifications and conformance between two specifications and implementation can be highly justified.

Explanations help bridge the gap between a system's formal representation and the understanding of application designers, who may be more accustomed to informal or practical perspectives. Providing explanations allows application designers to better grasp the system's behavior, requirements, and impact of deviations.

In the context of Imune, we explain as-Intended behavior and conformance between specification and implementation. While it is expected to provide explanations in natural language, other forms of explanation, such as visualizations or diagrams, can also effectively convey information. The current version of Imune still needs to support explanation in the form of natural language. Section 5.3 evaluates how well explained assists application designers in understanding the behavior of third-party components.

4.4.1. Explaining as-Intended Behavior

Summarizing Variables Domain Values. Variables are an essential part of specifications based on transition system formalism. The following are the main reasons why variables are essential.

- **State representation.** Variables represent system states by modeling essential aspects of the system.
- **State transition.** Variables specify how the system's state changes over time. Actions and operations modify the variable values, allowing

to model the behavior and dynamics of the system.

- **Verification.** Model checking tools like TLC examine the possible states and transitions between them by computing values for each variable.

Hence, end-users of third-party components need to understand the purpose of each variable in design specifications. Part of the explanation of as-intended behavior is summarizing a list of all variables and a set of possible values each variable takes, computed by a model-checking tool.

Concerning the Circuit Breaker example, Table 4.1 shows the summary for the two variables *circuitBreaker* and *calledInState*. When values are inside curly brackets {}, the variable takes a set of values. ⟨ ⟩ shows that a variable takes a sequence of values. Both are important for the correct interpretation of values. The variable *circuitBreaker* takes three values, either *closed*, *opened*, or *half*, which are computed by the TLC model-checking tool.

The variable *calledInState* takes a sequence of values. The expected interpretations are that (1) the maximum number of calls computed by the model-checking is five and (2) the maximum number of test calls allowed in a half-open state is 2. Note that maximum values are usually not the limitation of the specification, but the state space is restricted by constraints such as $Len(calledInState) \leq 5$.

Summarizing State Transitions. Variables are one aspect of the transition system formalism, while actions represent another. It is often the case that specifications involve multiple variables and numerous state changes, which can significantly complicate understanding the system's behavior.

The idea behind a summary of step-wise transitions between variables is that application designers can focus on the key transitions and comprehend the behavior of one or more variables without getting overwhelmed by the system's state-space complexity. For example, Table 4.2 summarizes possible value changes for the *circuitBreaker*. Table 4.3 visualizes step-wise changes for the *circuitBreaker* and *calledInState* together.

Variable	Set of values {} - Sequence of values ⟨ ⟩
circuitBreaker	half closed opened
calledInState	$\langle\ \rangle$ $\langle\ closed\ \rangle$ $\langle\ closed, closed\ \rangle$ $\langle\ closed, closed, closed\ \rangle$ $\langle\ closed, closed, closed, closed\ \rangle$ $\langle\ closed, closed, closed, closed, closed\ \rangle$ $\langle\ closed, closed, closed, closed, opened\ \rangle$ $\langle\ closed, closed, closed, closed, half\ \rangle$ $\langle\ half\ \rangle$ $\langle\ half, half\ \rangle$

Table 4.1.: Visualization of Circuit Breaker specification variables summary - only two of the entire variables are given.

Variables	The Set of Intended Step-Wise Values Change; From → To
circuitBreaker	closed → closed half → half opened → opened closed → opened opened → half half → closed

Table 4.2.: Visualization of Circuit Breaker step-wise value change - for one variable.

4.4.2. Explaining Conformance

Understanding conformance results plays an essential role in ensuring the correctness of systems. In particular, in the context of Imune, understanding conformance enables validating that observed behavior aligns with the specified behavior and, if not, how the observed behavior is different, what the potential reasons are, and what the impact is.

As a solution, Imune explains conformance in the form of a graph. The

Variables	The Set of Intended Step-Wise Values Change; From → To
[circuitBreaker, calledInState]	half, ⟨ ⟩ → half, ⟨ ⟩ half, ⟨ ⟩ → half, ⟨ $half$ ⟩ half, ⟨ ⟩ → half, ⟨ $half, half$ ⟩ half, ⟨ $half, half$ ⟩ → opened, ⟨ ⟩ half, ⟨ $half, half$ ⟩ → closed, ⟨ ⟩

Table 4.3.: Part of visualization of Circuit Breaker step-wise value change - for two variables.

graph represents how the behaviors align or not in terms of:

- **Conforms**: The next observed state is marked as conforms in two cases. First, the next observed state is the first conforming state in the generated conformance graph, which *only* requires the same set of variables with equal values to a state in the design specification. Second, when not first anymore, the next observed state must relate to the next step transition in the design specification, which means the observed state conforms if and only if the *next step* transition in the design specification arrives at a state with equal variables value to the observed state.
- **Diverges**: When the next observed state is not a next-step transition according to the design specification but reachable with more than two steps, the observed state is marked as diverged, indicating that one or more states are expected to be seen from the system runtime behavior but is *missing*.
- **Missing**: Represents states that are not observed from the system runtime but are expected according to the next-step transition in the design specification.
- **New**: Indicates two cases. First, the observed state is unreachable from previous conforming states, including green or yellow-marked states. Second, the observed state contains variables with values not equal to one in the design specification.

4.5. Assumptions and Limitations

This Section lists the most significant assumptions and limitations in Imune and the supporting tool. Concerning the state variable mapping model:

- **Auxiliary Variables.** Imune's support is currently limited to only the history variables. Values of the history variables are defined based on the current state and the past behavior, which allows reasoning about behaviors from the past until the current state. On the other hand, prophecy variables are typically used to predict future behavior. Typical use cases are recovery or role back operations [Aba15]. The prophecy variable is introduced to predict the effect of recovery, as the effect has not been observed yet.
- **Semantic Alignment.** It is assumed that mapped variables represent the same concepts and meaning. Otherwise, the resulting conformance checking is meaningless.
- **Compatibility.** It is assumed that a refinement mapping exists between the design specification variables and variables at the observable level.
- **Completeness.** The assumption is that necessary state variables are selected and included in the state variable mapping model.
- **Subjectivity.** Mapping state variables may involve subjective judgment or interpretation, as different individuals or teams may have different perspectives or understandings of the system.
- **Static Mapping Model.** It is assumed that the representation and meaning of state variables in both specification and observable do not change over time. If so, manual adjustments are needed.
- **No Support for Specified Properties.** Imune does not yet provide a means for including specified system properties in TLA$^+$ specifications. Related to properties is how identified deviations impact the specified properties.

Concerning the explanation of both as-Intended behavior and the conformance checking results:

- **No Support for Actionable Insights.** Currently, Imune does not provide suggestions on how deviations can be mitigated.

Chapter 5

Validation

This chapter presents the evaluation plan and results concerning the effectiveness of the Imune approach, including (1) correctness and (2) understandability concerning system behavior. Correctness refers to the ability to detect deviations. Although detecting deviations is essential, it is equally important for us to assess understandability by checking how well Imune users interpret Imune outputs, including specified behavior and deviations. Such understanding is often a precondition for re-configuration or communication with third-party code designers concerning deviations. The rest of this introductory text summarizes what is explained throughout the chapter.

The software projects for the tasks in evaluation are Producer-Consumer and ReadWrite lock patterns (Section 5.1). The following are the criteria for choosing the software projects. First, a third party writes the code and its design specification. Here, a third party refers to an organization or a person besides the author of this dissertation. Second, cover two types of failures according to the system failure definition from Avizienis et al., [ALRL04]. Systems failure is associated with *incorrect behavior* visible to the system environment. The incorrect behavior is because either the implementation *does not conform to its specification* or the specification *does not adequately*

describe the functionality. Both projects comply with the criterion.

One aspect of Imune's effectiveness is finding deviations between as-implemented and as-intended architecture. Hence, the Imune approach must possess the ability to detect deviations/failures. To show that, the author of this dissertation instantiates the Imune instance for both software projects written by third parties. Section 5.2 provides a more detailed explanation. In summary, concerning the producer-consumer pattern, Imune correctly identifies that observed behavior reaches a deadlock state and marks the state as non-conforming to as-intended behavior. Concerning ReadWrite lock, the specification inadequately specifies downgrading from *write lock* to *WriteRead*. The implementation follows the inadequate specification and allows another task to obtain either *Read* or *Write* lock while a task already holding exclusive *WriteRead*, which is not the true intention of the pattern. Imune correctly marks deviations and conformance of the downgrading state to its specification.

We conducted a comparative analysis experiment involving users to assess understandability. The objective is to compare Imune's effectiveness in understanding system behavior with an established code inspection approach. We chose code inspection because it's a well-established practice for understanding system behavior and identifying deviations between the as-designed architecture (often documented in comments) and the as-implemented architecture (the executable code). It's worth noting that reviewing observable metrics and logs is also common. However, comparing Imune with reviewing observables is challenging. Metrics only provide indicators of something happening and often lack deep insights into system behavior. While logs offer valuable information, they can be cumbersome to read without suitable visualization or log correlation techniques.

The assessment relies on quiz scores. Two tasks concerning third-party implementations of Producer-Consumer and ReadWrite lock design patterns are included in the experiment. Two separate groups utilize each method and complete quizzes tailored to each project. Each group consists of 5 participants, resulting in 10 data points for the understandability score. While the permutation test indicates no statistically significant difference

between the two methods, participants using Imune scored, on average, 26.35% higher than those using the code inspection method. Notably, in the Producer-Consumer task alone, participants scored, on average, 50.18% higher with Imune compared to code inspection, which suggests Imune's superior performance in understanding the behavior of a relatively more complex third-party component (the rationale for complexity is detailed in the Results Section 5.3.3).

5.1. Software Projects for Validating Imune

Evaluation is done in the context of two software projects, Producer-Consumer and ReadWrite lock. Both projects are designed and implemented by a third-party organization unaware of Imune. The following gives information about design specifications and implementation artifacts used for Imune validation.

The ReadWrite lock pattern is formally specified in TLA$^+$ and implemented in Python by independent third parties Niederbühl [Ale21] and aio-libs [aio22]. Regarding the producer-consumer project, the design specification is written in TLA$^+$ by Michel Charpentier and published under the title *Debugging Concurrent Systems with a Model Checker* as a blog post in *Level Up Coding* website [Mic21]. Markus Kuppe wrote the implementation code, publicly available in GitHub [Mar22].

According to the fundamental concepts of dependability [ALRL04], systems failure is associated with *incorrect behavior* visible to the system environment. The incorrect behavior is because either the implementation *does not conform to its specification* or the specification *does not adequately describe* the functionality. The Producer-Consumer project covers the first, and the ReadWrite lock project covers the second. The following describes the design specification and implementation of each project.

5.1.1. Producer-Consumer Description

The Producer-Consumer pattern prescribes a shared buffer between producers and consumers. It is a solution to the classical synchronization problem

between processes.

The parameters of the design specification are five producers, two consumers, and a buffer with a capacity of two. A producer waits when the buffer is full, and a consumer waits when the buffer is empty. Producers and consumers waiting are added into a set called *waitSet*. After a producer, p_1, produces data, the producer p_1 signals *awakes all* the waiting processes. The TLC model checker shows that the specification concerning the parameters and described behavior is deadlock-free, meaning that all producers and consumers cannot wait simultaneously.

The Java implementation with the same parameters has the possibility of deadlock, and during runtime, the system eventually goes into a deadlock where all producers and consumers wait. The written Java program uses *notify* method from *java.lang.Object* class to wake up processes that are waiting. According to the API documentation, the method "Wakes up a single thread waiting on this object's monitor. If any threads await this object, one is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation. A thread waits on an object's monitor by calling one of the wait methods."

When producers and consumers are both waiting in the same set, a critical error occurs where all producers and consumers eventually end up waiting. This error results in the failure of the implementation to comply with the specification. The issue arises because, upon data production by one producer, another producer might be notified instead of a consumer, leading to a deadlock where all parties are waiting indefinitely. For instance, imagine an empty buffer with the current waiting set of p_1, p_2, c_1, c_2. If process p_4 produces data and notifies another process, the Java API's notify implementation might arbitrarily select p_1, triggering another data production. As a result, when the buffer reaches full capacity, the notify API may arbitrarily choose p_2, causing it to wait. This waiting cascade ensues, resulting in a deadlock. Potential solutions include either utilizing Java API's NotifyAll or modeling separate waiting sets for producers and consumers.

However, it has to be noted that using the TLC model checker, it is inferred [1] that under the following condition, the system is deadlock-free when the *notify* method is used.

$$2 \times BufCapacity \geq Cardinality(Producers \cup Consumers)$$

5.1.2. ReadWrite Lock Description

Read-Write lock holds a pair of locks for a shared resource. Multiple reader tasks may hold the read lock simultaneously so long as there are no writers. The write lock is exclusive. The JavaDoc[2] describes the pattern as "A ReadWriteLock maintains a pair of associated locks, one for read-only operations and one for writing. Multiple reader threads may hold The read lock simultaneously so long as there are no writers. The write lock is exclusive. [...]". Based on the JavaDoc, one of the few policies for Read-Write lock implementation is to decide whether or not to allow a task to downgrade from write lock to read lock, i.e., WriteRead. The benefit is that the same task immediately gains read access and does not have to wait. In the project, the downgrading case is allowed for a task.

The design specification is inadequate for the scenario: *When a task holds a WriteRead lock, other tasks are not allowed to hold either a read or a write lock.* The design specification allows a task to obtain a read or write lock while the other has a WriteRead lock. The specification contains two variables, *State* and *Lock*, for modeling the state of shared resource and the status of the lock, including *Waiting*, *Write*, *Read*, and *WriteRead*. For model checking, the final model is instantiated with two tasks with an initial lock status of *Waiting* for each task and the state of 0 for the shared resource, meaning that the resource is unlocked. Positive values, for the *State* of the resource, means read lock, and a negative value means write lock.

[1] https://github.com/lemmy/BlockingQueue/tree/v08_continue

[2] https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReadWriteLock.html

5.2. Correctness

Imune must indicate conforming and non-conforming behaviors observed from the system at runtime compared to the system specification. This section presents Imune's correctness by conducting two experiments on the Producer-Consumer and Read-Write lock projects. The first project, Producer-Consumer, is to show that Imune correctly detects deviation by showing that there is a **new state** where originally was not intended according to the design specification. The second project, Read-Write lock, is to show that Imune correctly shows the system observed behavior aligns with the intended behavior. Although, the intended behavior does not adequately describe the system's function.

The rest details instantiating Imune for each project. Each of the Sections explains the following steps.

1. Characterizing specification state variables,
2. State variable mapping,
3. Comparison and visualization, and
4. Interpretation.

5.2.1. Producer-Consumer

Characterizing specification state variables. The system is deadlock-free According to the intended design specified in TLA$^+$. The specification has two variables *buffer* and *waitSet*, which model the shared storage between processes and the set of waiting processes. Formula 5.1 is an excerpt from the complete specification, available in the supplementary materials. A producer puts data into the buffer if the buffer has free capacity. On the other hand, a consumer consumes data whenever the buffer has something to read. After a producer p_1 puts data into the buffer, the producer notifies all the waiting

processes.

$$
\begin{aligned}
&NotifyAll \triangleq waitSet' = \{\} \\
&Wait(t) \triangleq waitSet' = waitSet \cup t \\
&put(t, m) \triangleq IF\ Len(buffer) < BufCapacity\ THEN \\
&\qquad\qquad buffer' = Append(buffer, m) \wedge NotifyAll \\
&\qquad ELSE\ wait \\
&Get(t) \triangleq IF\ Len(buffer) < 0\ THEN \\
&\qquad\qquad buffer' = Tail(buffer) \wedge NotifyAll \\
&\quad ELSE\ wait
\end{aligned}
\tag{5.1}
$$

The specification model has the following parameters and their values. In addition, the TLC model checker shows that the model does not reach a deadlock.

- Producers parameter has $\{p_1, p_2, p_3, p_4, p_5\}$ as the model value,
- Consumers parameter has $\{c_1, c_2\}$ as the model value, and
- BufCapacity is 2.

The mapping involves (1) identifying corresponding state variables in the source code, (2) instrumenting the code, and (3) mapping state variables through pattern matching. The specification has two variables, *waitSet* and *buffer*. The current implementation does not provide observable data related to the variables. In the context of this project, the code size is relatively small, and by code reviewing, the author of this dissertation identified the corresponding construct in the source code. Regarding the **waitSet**, a process is implemented as *Callable* from Java concurrency API. The variable *buffer* implementation is a Java class *BlockingQueue*.

```
public final class BlockingQueue<E>
```

One of the class's attributes is `E[] store;` that models the data inside the buffer. To observe the values of both variables, the author of this dissertation instrumented the program. The program runs with the same parameters

as the design specification. The excerpt of the observed data after running the program is

```
INFO: Buffer full; P: producer-4 waits
Method: getWaitSet,Return: [producer-3, producer-4]
Method: getBufferedValues,Return: producer-4,producer-2
...
INFO: C: consumer-1 running.
Method: getWaitSet ,Return: [producer-3, producer-4]
Method: getBufferedValues ,Return: producer-2
...
INFO: C consumer-1 took producer-4
INFO: P: producer-3 running.
Method: getWaitSet,Return: [producer-4]
Method: getBufferedValues,Return: producer-3,producer-2
...
INFO: P put producer-3
```

The first line of the logs shows that producer-4 attempted to put data into a buffer. However, the buffer is full. The *waitSet* variable contains producer-3 and producer-4. Moreover, the contents in the buffer belong to the producer-4 and the producer-2. Later on, the consumer runs and takes data. In this case, the data is producer-4. The log shows that *waitSet* still has producer-3 and producer-4 at the time. Notifying one of the producers is not immediately visible as the consumer has to leave the synchronized method until one of the producers continues.

To map the observed variable to the variables at the design specification, the excerpt of the mapping model for the *waitSet* variable is:

```
kind: StateMapping
variables:
  - specName: waitSet
    isSequence: false
    isSet: true
    isHistory: false
    valueFromFields:
  - methodName: getWaitSet
    ...
mappingExpression:
  - ifCondition: "consumer-1"
    subtitute: "c1"
```

After letting the system run for around 20 minutes, the observable for waitSet has the following value {*producer-1, producer-2, consumer-2, producer-5, consumer-1, producer-3, producer-4*}, which means the system has reached a deadlock. Figure 5.1 shows the trace inclusion check and the system execution path, including deviations.

The previous steps produce two configuration files necessary for the next steps, including comparison and visualization.

Comparison and visualization. This step is automated and uses previous configuration files. The comparison algorithm and visualization are based on the trace inclusion checking and DOT graph described in the Imune chapter. Figure 5.1 shows the output from Imune tooling. The graph shows a sequence of implementation states observed at runtime.

Interpretation. The initial observed state is where variables *waitSet* and *buffer* take values $\{p_1, p_2, p_3, p_4\}$ and *data*. The state is marked as new because, according to the reachability graph generated from the design specification, there is no state where producers are in a waiting set as long as the buffer capacity is not full. Those states with full buffer capacity do not exist. According to the reachability graph, a consumer waits only for an empty buffer. The final observed state shows that deadlock is reached as all

the producers and consumers are waiting.

5.2.2. Read-Write Lock

Characterizing specification state variables. The design specification must guarantee that while a task $task_1$ holds a *WriteRead* lock on a shared resource, another task $task_2$ does not get read or write access. The specification inadequately describes the pattern, leading to the failure to hold the guarantee. The specification has two variables, *Lock* and *State*, modeling the status of tasks and the shared resource, respectively. The *Lock* is a function with a set of tasks as the domain and the task status as the range. The domain is parameterized with $\{task_1, task_2\}$ as the model value and range is an element of the set $\{$"*Read*","*Write*","*WriteRead*","*Waiting*","*Finished*"$\}$. The state is an integer value. $State > 0$ shows that the shared resource has a read lock, while $State = 0$ and $State < 0$ show that the resource is unlocked and has a writing lock, respectively.

Formula 5.2 is a segment of TLA$^+$ specification for Read-Write lock. The segment specifies how a task t acquires a read lock through the *RAquire* action. The action has two enabling conditions. A task acquires a read lock if (1) another task does not have a write lock on that resource, which is specified through the operator $Wlocked \triangleq State < 0$, or (2) the same task t has already a write lock on the resource, which is specified through the operator $WOwn(t) \triangleq Lock[t] \in$ "$Write$". The formula shows allowance for downgrading from *Write* to *WriteRead*.

Formula 5.3 specified write acquire action. The action is enabled when the state is unlocked, modeled as the operator $Unlocked \triangleq State = 0$.

Inadequate specification of the Write-Read lock is visible in the two Formulas. When a task acquires a write lock, the value of *State* variable becomes -1. A later downgrade to WriteRead makes the state of resource 0, which means the resource is now unlocked, and other tasks can obtain a read or a

write lock.

$$
\begin{aligned}
RAquire(t) \triangleq\ & \neg Wlocked \\
& \land\ Lock[t] \in \{"Waiting"\} \\
& \land\ Lock' = [Lock\ EXCEPT\ ![t] = "Read"] \\
& \land State' = State + 1 \\
& \lor\ \land\ WOwn(t) \\
& \land\ Lock' = [Lock\ EXCEPT\ ![t] = "WriteRead"] \\
& \land\ State' = State + 1
\end{aligned}
\tag{5.2}
$$

$$
\begin{aligned}
WAquire(t) \triangleq\ & Unlocked \\
& \land\ Lock[t] \in \{"Waiting"\} \\
& \land\ Lock' = [LockEXCEPT![t] = "Write"] \\
& \land State' = State - 1
\end{aligned}
\tag{5.3}
$$

State variable mapping process. The mapping process involves (1) identifying corresponding state variables in the source code, (2) instrumenting the code, and (3) mapping state variables through pattern matching.

The specification contains two variables *Lock* and *State*. At the implementation level, the state is modeled as an integer. In contrast, *Lock* is not directly implemented and must be inferred from existing observables. For example, to observe the values of the variable *Lock*, the following code is added before two tasks are executed. The status must be set accordingly when a task obtains the requested lock.

```
lockStatus = { "task1":"Waiting","task2":"Waiting"}
```

Regarding recording behavior at runtime, There are three test cases for three scenarios of two tasks attempting to gain a lock on the shared resource, including (1) Read and Read, (2) Write and Read, and (3) WriteRead and Write. The goal is to cover various system behavior concerning the system

functionality. The excerpt of the observed data after running the program is given as follows.

```
Method: getWriteLockState,Return: <WriterLock: [unlocked]>
Method: getReaderLockState,Return: <ReaderLock: [unlocked]>
Method: getState,Return: 0
Method: getTasksStatus,Return:
{'task1': 'Waiting', 'task2': 'Waiting'}
Task 1: acquiring write lock;
Task 1: acquired write lock;
Method: getState,Return: -1
Method: getWriteLockState,Return: <WriterLock: [locked]>
Method: getReaderLockState,Return: <ReaderLock: [unlocked]>
Method: getTasksStatus,Return:
{'task1': 'Write', 'task2': 'Waiting'}
Task 1: downgraded to read lock;
Task 1 is in WriteRead mode;
Method: getWriteLockState,Return: <WriterLock: [unlocked]>
Method: getReaderLockState,Return: <ReaderLock: [unlocked]>
Method: getState,Return: 0
Method: getTasksStatus,Return:
{'task1': 'WriteRead', 'task2': 'Waiting'}
```

The state variable mapping configuration for the variable *Lock* is as follows.

```
kind: StateMapping
variables:
  - specName: State
    isSequence: false
    isSet: false
    isHistory: false
    function:
    valueFromFields:
      - methodName: getState
          ...
    mappingExpression:
      - ifCondition: -?\d+
        subtitute:
```

Comparison and visualization. The DOT graph for the Read and Read test case is visible in Figure 5.2. The graph shows that the first observed system state is where the shared resource is unlocked, i.e., the *State* variable equals zero, and both tasks are waiting. The node conforms, i.e., a state exists in the reachability graph with the same variables taking equal values. The next state transition is where the variable value of *State* is 2, and the status of one task shows that the task took *Read* lock while the other is still waiting. The task is non-conforming as the state does not exist in the reachability graph. The reason for observing the state is that both tasks took the read lock. However, the second task is getting executed with a slight delay, and the code statements that update the status for task 2 are not yet executed.

The next state transition is marked as diverged. According to the trace inclusion algorithm, reaching The state marked as yellow in *one-step* from the initial conforming state is impossible. The missing state between the previous conforming and diverged state is visible in the graph. The current conforming state is set to the yellow-marked state. The next states conform and show that each task released the lock and finished processing.

Figure 5.3 shows the third test case about the downgrading scenario. The last five states are non-conforming. Note that the fourth state from the

sequence path conforms and shows that the state of the shared resource is unlocked while one task has *WriteRead* and the other is still waiting. The next state transition shows that the *task1* releases the read lock, and the status of the shared resource is increased by one, which is not as intended according to the specification. Read release results in decrement according to the design specification. All the next states are marked as non-conforming as each is not reachable from the previous conforming state according to the reachability graph.

Interpretation. The last conforming state in Figure 5.3 shows that the intended design inadequately described the downgrading action as the shared resource is currently unlocked, which means the other task can gain either read or write lock.

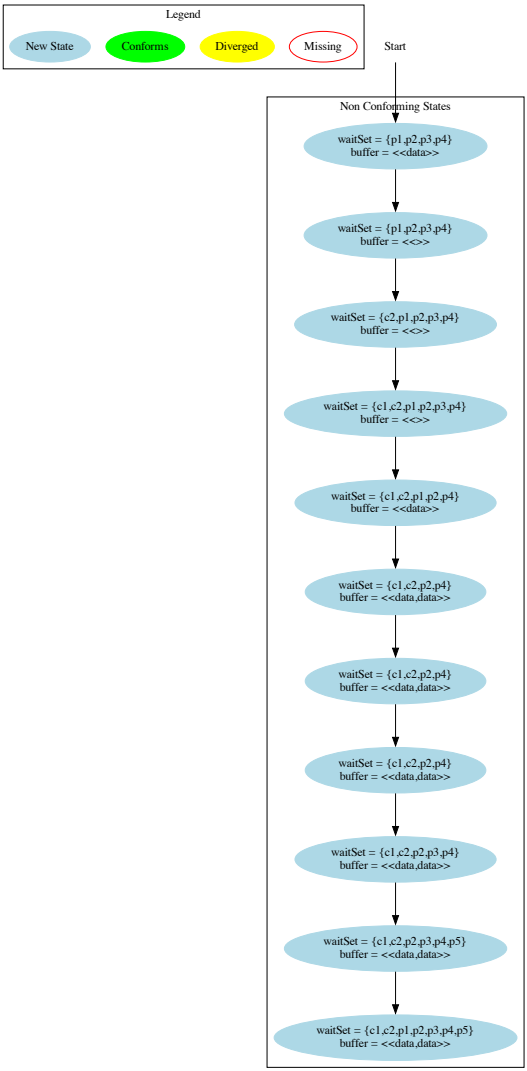

Figure 5.1.: Generated DOT graph after trace inclusion check. Output from Imune tooling

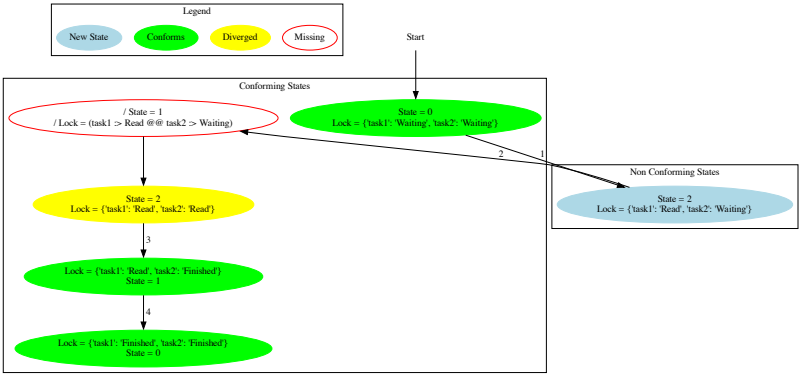

Figure 5.2.: Generated DOT graph after trace inclusion check. Output from Imune tooling

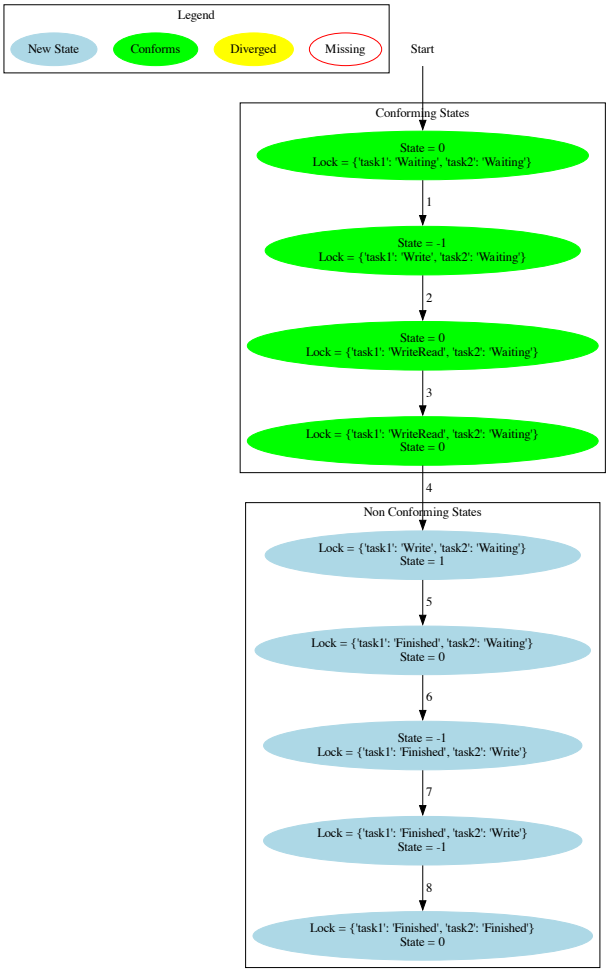

Figure 5.3.: Generated DOT graph after trace inclusion check. Output from Imune tooling

5.3. Understandability User Study Experiment

To assess understandability, we follow an experiment process visible in Figure 5.4. The first step includes defining goals, questions, and related hypotheses. The second step is about designing the experiment. In particular, making decisions about

- Within-subjects versus between-subjects studies,
- Training,
- Quizzes for understandability assessment,
- Online user study versus on-site, and
- Expected time bound for finishing the tasks.

The third step concerns the actual execution of the experiment. The fourth and fifth steps concern the interpenetration of the collected data and documentation, respectively.

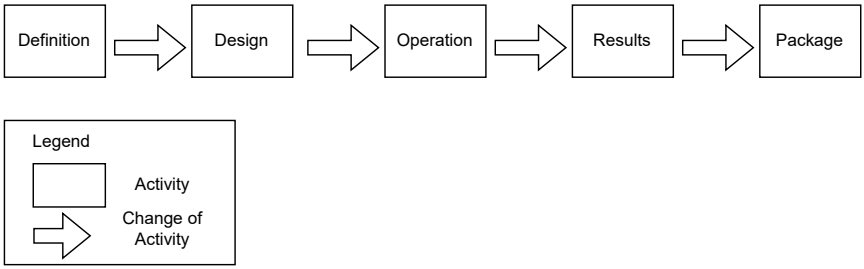

Figure 5.4.: Experiment Process based on [WRH+12]

5.3.1. Definition

One objective behind the Imune method is to enhance application designers' understanding of the behavior of third-party components. To enhance understandability, a base approach for comparison is required. In this regard,

the choice is code inspection. While both methods, Imune and code inspection, target behavior understanding, there are challenges in performing a reasonable comparison between them. Here are some reasons.

1. **Understanding Scope:**

   code inspection. Focuses on identifying bugs, vulnerabilities, coding standards, and implementation details such as modules, algorithms, and data structure.

   Imune. Concentrates on runtime behavior, aiming to detect deviations and unexpected behavior at runtime compared to the Specified behavior.

2. **Coverage:**

   code inspection. Gives detailed insight into module structure and logic and catches coding mistakes, which often is not the purpose of behavioral conformance checking.

   Imune. Captures issues related to unintended behavior at the design level.

3. **Automation:**

   code inspection. Is done manually by humans to identify bugs and errors

   Imune. Leverages automated solutions for comparison between as-designed and as-implemented

Hence, designing an experiment that compares the two requires careful consideration. We address the scope concern by identifying potential system failure, not detailed logic, and uncovering bugs (s). Identifying bugs based on Imune outputs is currently not supported, as there is no link between the deviations and code statements that have been executed. Regarding coverage, the study design considers that participants' answers must be corrected differently as gaining insight is at a different abstraction level. Automation impacts the time it takes to finish a task for the study. Hence,

the code size is chosen to be relatively small, which makes the study fair regarding automation.

Based on the idea of comparing two behavior understanding methods, the experiment definition is following. The structure of the experiment definition is based on the experimentation in software engineering book [WRH+12].

Object of Study Analyzing third-party components behavior understanding methods.

Purpose To evaluate Imune in comparison to code inspection.

Quality The quality focus is effectiveness in terms of a score for quizzes and time to finish each behavior understanding task.

Perspective From the perspective of third-party components end-users, including developers or operators.

Context The context is two software projects (1) Producer-Consumer and (2) Read-Write lock patterns. The participants are characterized as individuals, which means there is no teamwork. Moreover, the required code inspection experience is basic Java and Python knowledge. For Imune, previous knowledge of state-space modeling is sufficient.

5.3.2. Design

This Section presents the decisions behind the planning of the user study experiment, including the

- Hypotheses,
- Independent and dependent Variables,
- Choice of participants,
- Within-subjects vs. between-subjects study,
- Training,
- Quizzes for understandability assessment,
- Synchronous user study versus Asynchronous, and

- Expected time bound for finishing the experiments.

Regarding the time-bound, estimating the time required to finish each task using Imune and code inspection is essential. Time-bound allocation is justified because there is no desire in both practice and academia to solve tasks after dedicating an infinite amount of time. Based on results from pilot runs, we estimated 1 hour for each task regarding code inspection and Imune.

In addition to questions related to this experiment's hypotheses, it is interesting to know the complexity of the Imune method and the tooling from the user perspective. We surveyed participants' subjective reactions to using the Imune method. The survey contains a reduced set of exact similar items proposed by brooke [Bro+96] for usability assessment of tooling in the industrial context. A reduced set was mainly to keep the survey short as participants finished two tasks and were most likely tired of concentrating on filling out a lengthy survey. The set of items is a Likert scale of 5 points, and participants indicate the degree of agreement or disagreement, except for the last item, which is an open-ended question.

Hypotheses. To test whether Imune outperforms code inspection, metrics and hypotheses testing is needed. We designed the same set of quizzes for Imune and code inspection. The assessment of understandability is based on two factors. First, the score that participants get from the quiz. The rationale is that in practice and academia, quizzes are used to assess the level of understanding. For example, in academia, written exams are a means for assessing students. The rationale is further justified by other works performing user studies to assess understandability using quizzes such as an understandable consensus algorithm research [OO14].

The second factor is the time to finish each experiment. Often, in academia and practice, time to finish a quiz must be bounded. The main questions and related hypotheses are based on the two factors.

1. Does Imune outperform the code inspection method concerning understandability? How different is the understandability score between the two approaches, Imune and code inspection?

   The hypothesis is that Imune users score higher. Our reason is that Imune outputs precise as-Intended behavior and a runtime trace represented as a conformance graph. In contrast, users of code inspection rely on comments and have to construct system runtime execution model in their minds. However, it has to be mentioned that our expectation based on the pilot runs is not a large effect size. If there is a significant difference, we expect the effect size to be 0.2.

   Here is some additional context about experiment preparation. The last question in each quiz asks participants to justify the possibility of a system failure. Imune detects the error that leads to failure. For instance, in the Producer-Consumer, Imune detects that all processes are waiting. For a fair comparison, the runtime observable that contains the error is removed from both tasks. Hence, Imune users do not see the failures in the graph.

2. Does Imune make users finish their understandability tasks faster than code inspection?

   We hypothesize that by using Imune, the time to finish each experiment is lower compared to code inspection. Our reasons are two. First, reading and understanding natural language is more time-consuming than the precise explanations provided in Imune. Second is that users of Imune do not have to execute code in their minds and see the specification and runtime traces.

Below are the hypotheses related to the complexity assessment using the survey.

1. I imagine most people would learn to use the system very quickly.

   The hypothesis is that most tend to agree with the statement, mainly due to the relatively low number of functionalities and the given initial training.

2. I found the system very cumbersome to use.

   The hypothesis is that the majority agree with the statement because Imune provides a Command-Line interface.

3. I think that I would need the support of a technical person to be able to use this system.

   The hypothesis is that the majority tend to disagree similarly to the first item due to relatively low functionality and initial training.

4. I needed to learn a lot of things before I could get going with the system.

   The hypothesis is that most participants tend to agree with the statement. The reason is to introduce the concepts of conformance algorithm, model checking, and reachability graph.

5. Do you have any additional comments?

Variables. The independent variables are

- Third-party components behavior understanding method, and
- Participants' experience.

Additionally, the order in which the tasks are solved is another independent variable. After solving the first task, participants get more familiar with the type of questions and activities they should do. Participants' experience with using code inspection and Imune is not identical. For code inspection, an experienced participant knows static analysis methods in general and, in particular, has been part of a code reviewing team before. An experienced user of Imune already knows the concepts of model-checking and monitoring systems, including instrumentation techniques and log analysis.

Within-subjects vs. between-subjects study. The experiment compares participants' understandability when using code inspection and Imune. A within-subject is disadvantageous because participants solve tasks using code inspection and then solve similar tasks using Imune. First, participants' responses to the tasks most likely get affected after going from one treatment to another that contains similar tasks with similar quizzes. Second, there is a chance that participants get tired and lose motivation after finishing all the tasks using one treatment. Hence, the experiment employs a between-subjects design with two different participant groups. One group uses code inspection to solve the tasks, and the other group uses Imune.

Choice of Participants. The main precondition we set for participation is that students should be upper-level bachelor students, meaning semester six and above. In addition, for code inspection, basic knowledge of both Java and Python was announced to be expected from participants. During the user study, unfortunately, the SQA group where the author of this dissertation works did not offer any courses. Hence, it became a challenge to have a large sample size. In total, there were 10 participants. Three of them finished their bachelor's studies. One PhD student and the rest either finished their master's or were in the end semester. Two participants who finished their master's were already working as a software developer and a DevOps engineer.

For code inspection, we chose participants more carefully concerning their programming skills. Not all participants may have experience in doing code reviewing during the experiment. However, all the participants had at least two years of programming experience—some as Hiwi students and some in academic projects.

Figure 5.5 illustrates each participant's distribution to the respective group. Note that in the code inspection, there are 6 participants. However, two bachelor students were already familiar with the producer-consumer example, meaning they had already listened to a lecture describing the implementation of producer-consumer and possible failure. Hence, their answers to the second task only are considered.

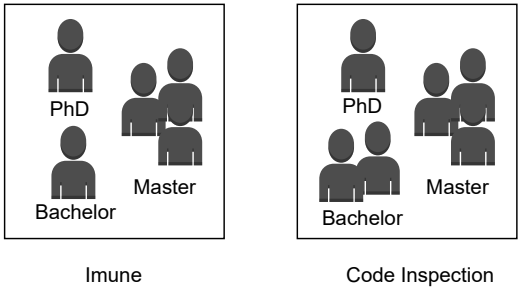

Figure 5.5.: Participants distribution to each group.

Training. For both code inspection and Imune, we provide an online repository for both materials and an explanation of the protocol. The protocol contains (1) initial phase, (2) running phase, and (3) final phase. Regarding the code experiment, the initial phase refers to the description of the division into two groups and the purpose of the study. The initial phase in Imune starts with an explanation of the Imune method, followed by the installation and usage of the tool. As an initial training, we provide a tutorial to explain the conformance graph, command options, and the outputs from the tool. In addition, we offer a template for both user studies to guide participants in recording their results and timing. The answers to questions are all open-ended. The initial phase in the Imune study is in supervised mode, meaning that participants go through the tutorial by themselves, and in case of issues, they can ask questions about anything, such as concepts implemented by tooling. During the running phase for both code inspection and Imune, the participants solve the tasks. In the final phase, the participants are guided to (1) upload their recorded results to a shared folder associated with their group and (2) fill out a usability survey.

Quiz. Designing quizzes and grading them is a way to assess the understandability of the participants concerning both as-designed and as-implemented behavior in third-party components. Quizzes are of three categories of difficulties, including easy, medium, and hard. Following is the description of the categories.

- Easy tasks require no step execution in mind and just looking at the comments, descriptions, or outputs.
- Medium tasks require one or two steps of execution in mind.
- Hard questions require keeping track of a few steps of execution in mind.

For grading, we thought of two options. The first is assigning points to each quiz based on the time it takes to solve it. However, there is one drawback to using such a grading rule. Both code inspection and Imune studies provide an exact set of questions that need to be solved using two different techniques. Hence, assigning points based on needed time makes less sense as the required time is expected to vary between different techniques. Our solution is designing our grading rubric. The criteria for assessment are (1) accuracy of answers, (2) providing evidence to support the answers, and (3) writing their answer using clear and concise language. It is possible to measure all three criteria by the provided answer. In addition, they are distinct from each other.

The accuracy of answers evaluates whether the participants accurately describe the intended or implemented behavior. Providing evidence criterion evaluates whether the answer is justified by example from given code or Imune outputs. Finally, clarity evaluates how precise and concisely participants provided their answers and justification. The assumption is that a good understanding usually leads to a more precise and concise answer, which is why the third criterion exists.

Our grading rubric is:

- **Accuracy (40%).**

  (0 points) There is nothing written at all, the answer is wrong, or there is some answer, but the connection to the question is hardly visible.

  (1-2 points) Incomplete answer, or some correct answer with minor errors

  (3 points) The answer is entirely correct.

- **Providing evidence (50%).**

  (0 points) Either there is nothing written at all, or the evidence is not based on the given code or Imune outputs

  (1-2 points) evidence is given but does not fully support the answer.

  (3 points) The evidence is given and fully supports the answer.

- **Clarity (10%).**

  (0 points) The language is very hard to understand

  (1-2 points) Somewhat clear with occasional language errors

  (3 points) Very clear and concise

Let us assume a participant scores *accuracy* with 3, *providing evidence* with two, and *clarity* with 3. The total score is $(3\times0.4)+(2\times0.5)+(3\times0.1)=2.5$. All the questions and possible solutions expected by the author of this dissertation is in the appendix.

Pilot Runs. As mentioned in the experimentation in software engineering book [WRH+12], it makes sense to have pilot runs for user studies to uncover critical issues in the settings, including designed quizzes, protocol, tasks, expected time, and possible technical issues. All the mentioned aspects

were essential for us as we could not tolerate the user study failure, mainly because finding participants is very difficult.

For Imune, we ran three pilot runs with four colleagues of author of this dissertation. The very first one uncovered a tooling bug. The second run provided feedback that it is not easy to comprehend the output without a tutorial explaining the Imune underlying concepts. Afterwards, there were two changes in the user studies' design. The first is designing a tutorial that helps participants to learn the tooling concepts, outputs, and command options. The second is replacing the Circuit Breaker project with the ReadWrite lock project. The main reason for replacing the Circuit Breaker project was the large code base of its implementation. Hystrix and Resilience4j implementations have large code bases, which does not allow a fair comparison between Imune and code inspection. We expected that reviewing a large code base could go beyond a day. Furthermore, participants require more skills in new Java features.

With the help of the final pilot run for Imune, we made final adjustments to the questions. In particular, we added more information for the description, which was missing and raised questions during solving tasks. After fixing the questions, we conducted a pilot run for code inspection with three participants.

Async mode. We decided to design the study in asynchronous (Async) mode. This approach involved sharing a repository containing all necessary materials and the study protocol, which outlines the study's purpose and the procedures participants should follow. Below, we list the reasons behind our choice. The main reason for opting for asynchronous mode was to motivate users to complete the study. Often, scheduling a dedicated location and time is inconvenient for participants. In our case, this was particularly challenging as each participant had different availability, necessitating multiple appointments for conducting the study.

5.3.3. Results

This Section presents the results obtained from the operation phase of the experiment.

Figure 5.6 compares the understandability score between Imune and code inspection. The maximum understandability score for the producer-consumer task is 18 as the number of questions is 6, and for ReadWrite lock is 30 as the number of questions is 10. The score of each participant for each task is normalized. For example, if a participant scores 30 for the ReadWrite lock task, the final score is 1. Looking at the results of each participant for both tasks, On average, participants scored 26.35% higher when using Imune compared to the code inspection method. The median score for code inspection is 0.73, whereas, for Imune, the median score is 0.90.

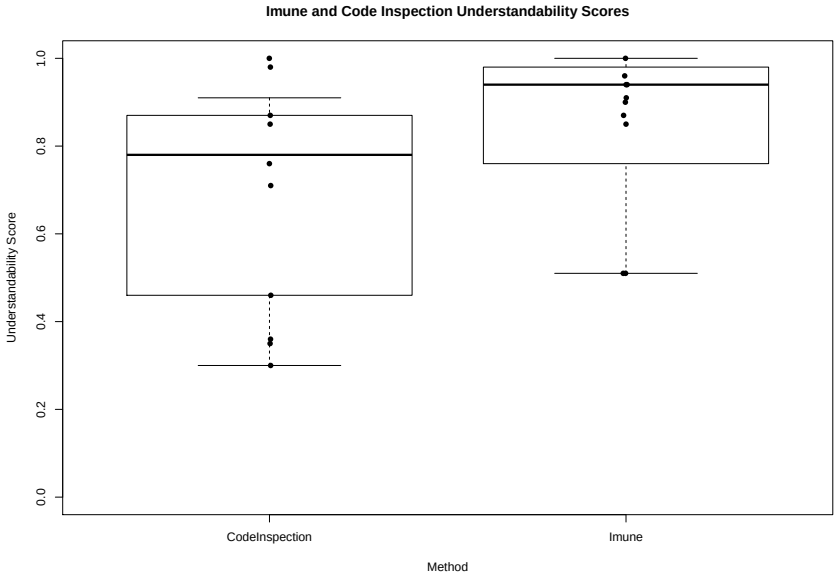

Figure 5.6.: Visualizing Imune and code inspection with a box-and-whisker plot. Emphasizing on median and outliers.

It is also interesting to look at participants' scores concerning individual tasks. Figure 5.7 compares participants' scores for only producer-consumer tasks. On average, participants score 50.18% higher using Imune compared to code inspection.

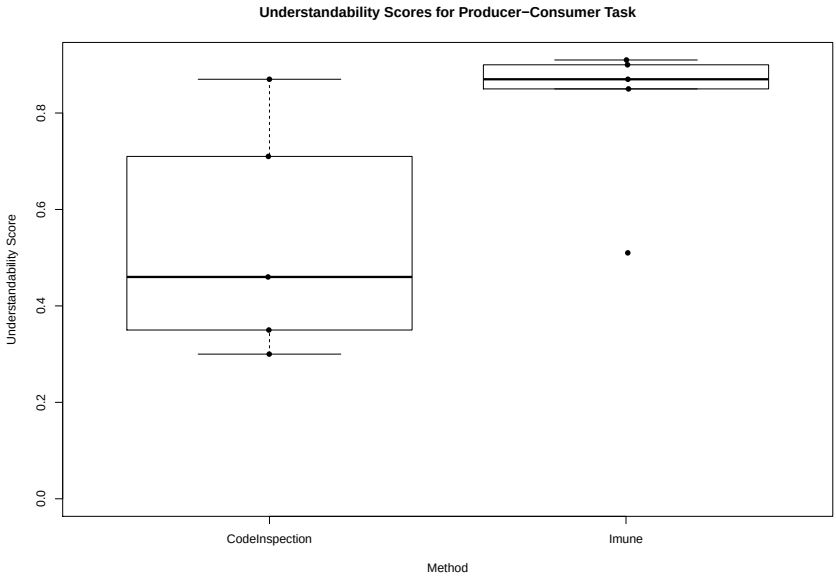

Figure 5.7.: A box-and-whisker plot for visualizing understandability scores concerning Producer-Consumer task.

The result can be justified because reading and understanding producer-consumer is complex, whereas Imune models the entire system using only two variables, and observable similarly involves two variables. Next, we list our subjective opinion of why the producer-consumer task is complex.

1. Multi-threading. The implementation involves multiple producer and consumer threads, each executing concurrently. Managing synchronization and communication between these threads is inherently complex.

2. Shared buffer. The producer and consumer threads interact through a shared buffer, which adds complexity due to potential race conditions when accessing and modifying the buffer.
3. Synchronization primitives. The implementation utilizes a lock to coordinate access to shared objects.
4. Thread coordination. The producer threads wait when the buffer is full, and the consumer threads wait when the buffer is empty. The implementation requires careful coordination using the condition variable to ensure the correct behavior of producers and consumers.

Figure 5.8 compares the understandability score for solving the ReadWrite lock task. On average, participants score 9.29% higher when using Imune than code inspection.

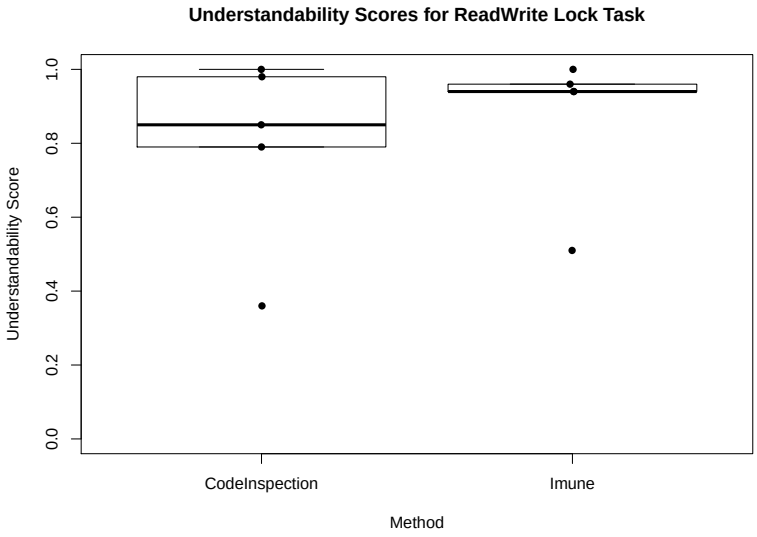

Figure 5.8.: A box-and-whisker plot for visualizing understandability scores concerning ReadWrite lock task.

The explanation is that the ReadWrite lock has a relatively simpler implementation than the producer-consumer task because ReadWrite lock implementation focuses on providing exclusive access for write operations and shared access for read operations using simple counters to manage the concurrent access to shared objects requirement.

Another result from looking at Figures 5.7 and 5.8 is that Imune shows a stable performance concerning understandability score compared to code inspection. The more considerable difference in median values measured by the code inspection method suggests variability in the method's effectiveness concerning the understandability score. The result can be justified by considering that participants using code inspection need to understand two implementations at different levels of complexity. In Imune, both tasks have specifications with two variables modeling the entire behavior and three to four log files for trace inclusion checking.

Figure 5.9 compares the time to finish the tasks between Imune and code inspection methods. On average, participants dedicated 50.1% more time to solving the tasks using Imune. The mean time to finish code inspection is 49.7 minutes, and the mean for Imune is 74.6 minutes. In addition, the permutation test for only the Producer-Consumer task confirms that code inspection outperforms Imune, i.e., the null hypothesis is rejected with the p-value of 0.02. However, the average for Imune is affected by two data points far from the center of the data set. Our justification for why Imune takes more time is that the method requires familiarity with concepts within formal methods, monitoring, as well as tooling, which initially may slow down the conformance checking process. Hence, maybe a learning curve is involved. On the other hand, code inspection only requires familiarity with programming languages.

The visualization of the recorded experience factor supports the justification. Figure 5.10 presents the independent variables of experience and time to finish measurements for code inspection and Imune. The objective is to identify possible correlations through visualization. As depicted in the visualization (plot on top), experience shows a weak correlation with time when using Imune. Participants with more experience tend to spend less

time completing the understandability tasks. In contrast, the correlation between experience and timing when using code inspection is much weaker, as observed in the bottom plot. Therefore, the insight gained is that Imune is more sensitive to experience. However, due to the small sample size, we consider the relationship between experience and time as an insight rather than a conclusive result.

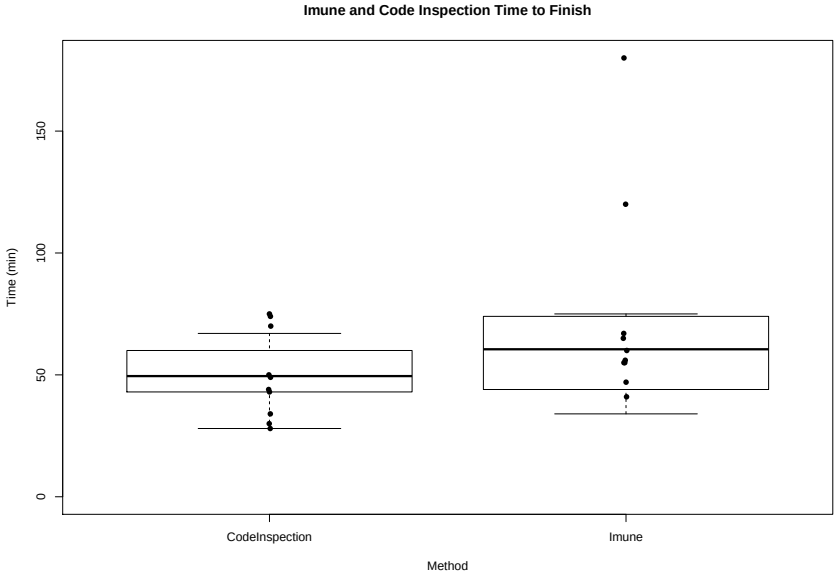

Figure 5.9.: Visualizing time to finish the tasks with a box-and-whisker plot. Emphasizing on median and outliers.

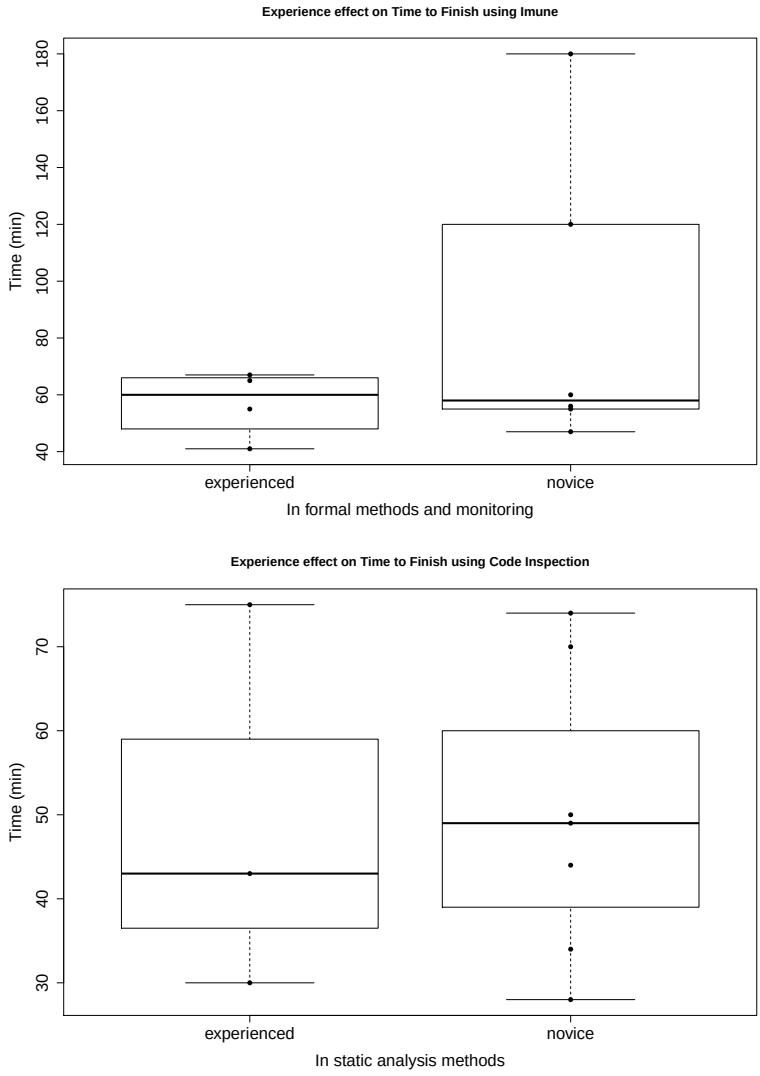

Figure 5.10.: Visualizing the relationship between experience levels and time to finish for Imune (above) and code inspection (bottom).

Figure 5.11 depicts the impact of experience on the understandability score. The bottom plot of the Figure shows a significant overlap between the boxplots, indicating that the data shows little difference in understandability scores between novices and experienced users. However, in the top plot for Imune, a negative correlation between experience level and understandability is evident, suggesting that more experienced participants scored lower than novices.

Our interpretation of this result is twofold. First, experienced users might assume certain aspects are self-evident and, therefore, do not provide explanations. Second, effective communication of understandability may not solely depend on experience but also on proficient communication skills, particularly in writing, which is significant in our case.

Tables 5.1, 5.3, 5.2 and 5.4 detail the understandability score and the corresponding time to finish based on the methods and experience factors. According to Table 5.1 the meantime to finish the Producer-Consumer task using code inspection is 40 minutes, while the mean for ReadWrite lock 5.2 using the same method is 59. Here are our three most important subjective reasons. First, concerning the ReadWrite lock, reading and understanding as-intended behavior became time-consuming as the description purposefully was not precise, and most likely, participants tried to repeat it a few times and understand the requirements of the implementation precisely. It must be noted that talking to a few participants also supports our subjective opinion. Second, participants do not often program in Python compared to Java. Hence, reading and understanding as-implemented behavior became time-consuming. Third, the ReadWrite lock task has four more questions than the Producer-Consumer task. However, the author's subjective opinion is that the third reason is less influential regarding time.

Participants dedicated less time to the complex task Producer-Consumer using code inspection. One possibility is the limitation of code inspection that mainly touches the surface level, such as identifying style issues or functional correctness within the class level. Participants most likely did not find suspicious interactions within code statements, and hence, it did not become necessary to investigate further and build a mental model of the

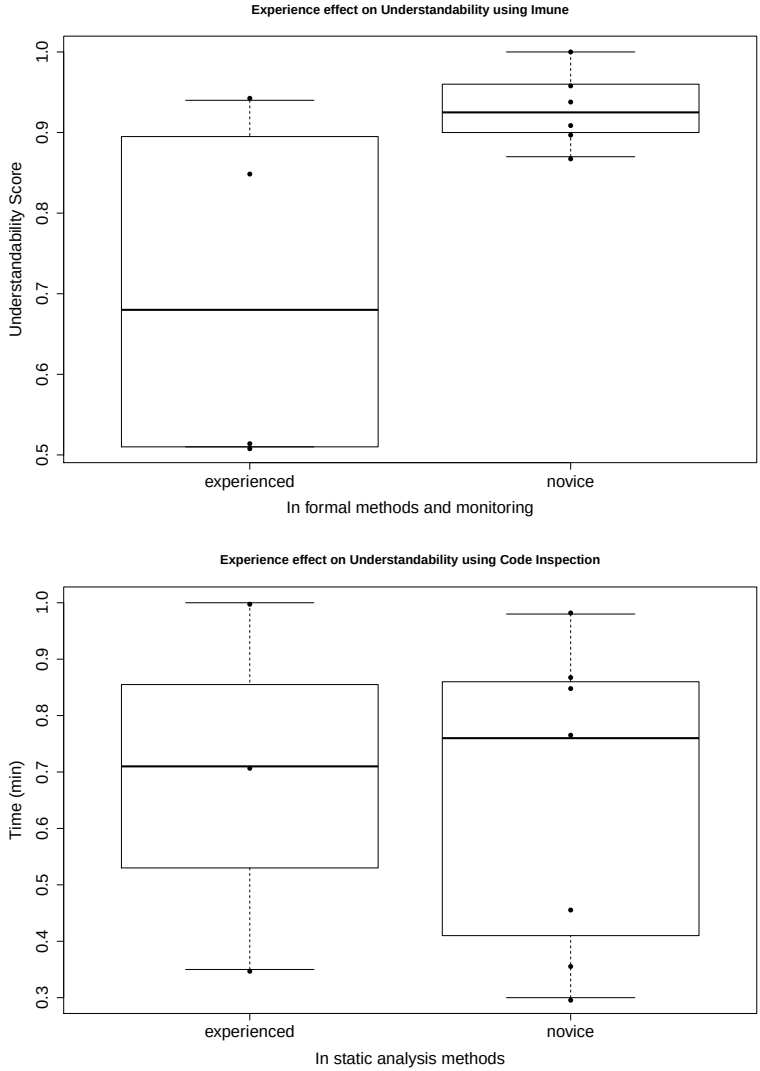

Figure 5.11.: Visualizing the relationship between experience levels and understandability for Imune (above) and code inspection (bottom).

system execution.

Participants	Score	Time (min)	Experienced
1	0.46	49	No
2	0.35	43	Yes
3	0.87	50	No
4	0.3	28	No
5	0.71	30	Yes

Table 5.1.: Measurements for the Producer-Consumer task using code inspection.

Participants	Score	Time (min)	Experienced
1	0.76	74	No
2	1	75	Yes
3	0.85	70	No
4	0.98	44	No
5	0.36	34	No

Table 5.2.: Measurements for the ReadWrite lock task using code inspection

Participants	Score	Time (min)	Experienced
1	0.9	60	No
2	0.87	47	No
3	0.51	67	Yes
4	0.91	180	No
5	0.85	55	Yes

Table 5.3.: Measurements for the Producer-Consumer task using Imune.

Participants	Score	Time (min)	Experienced
1	0.96	56	No
2	1	55	No
3	0.51	41	Yes
4	0.94	120	No
5	0.94	65	Yes

Table 5.4.: Measurements for the ReadWrite lock task using Imune.

Usability results. Participants of Imune filled out a survey aiming to collect measures for assessing Imune tooling usability in terms of users' subjective reactions to using the system. Next, provide the results for each item. The set of items are Likert scale of 5 points, and participants indicate the degree of agreement or disagreement. Figure 5.12 shows the result concerning the item *I think that I would need the support of a technical person to be able to use this system.* Figure 5.13 shows the result concerning the item *I imagine most people would learn to use the system very quickly.* Figure 5.14 shows the result concerning the item *I found the system very cumbersome to use.* Figure 5.15 shows the result concerning the item *I needed to learn many things before I could get going with the system.*

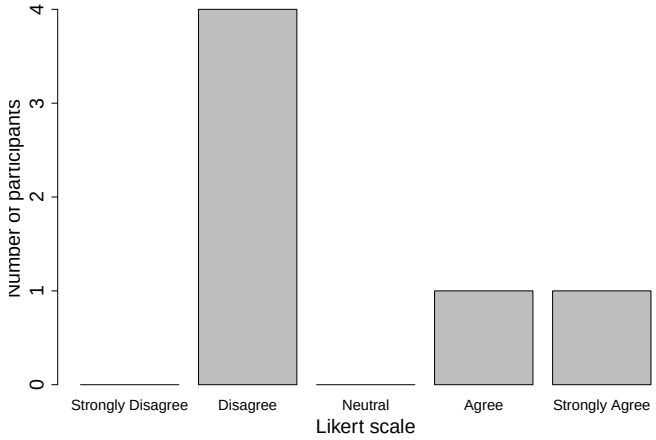

Figure 5.12.: Results for : I think that I would need the support of a technical person to be able to use this system.

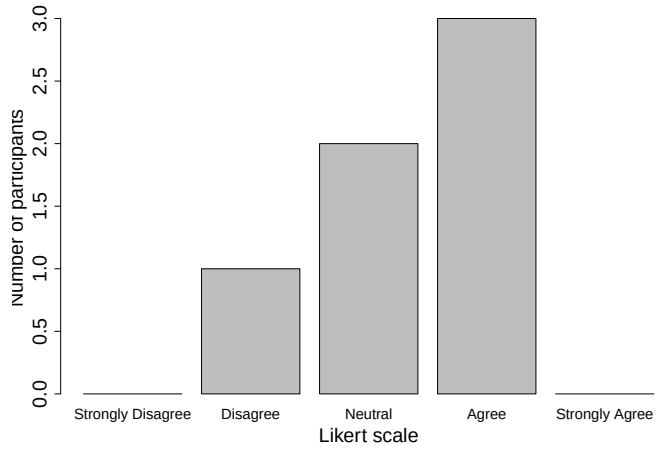

Figure 5.13.: Results for: I imagine most people would learn to use the system very quickly.

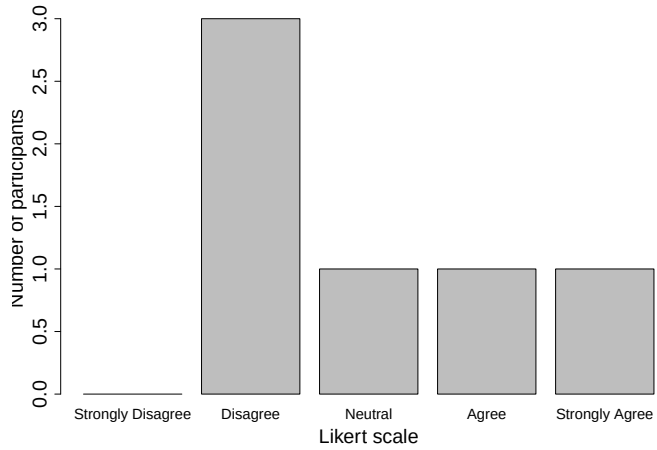

Figure 5.14.: Results for: I found the system very cumbersome to use.

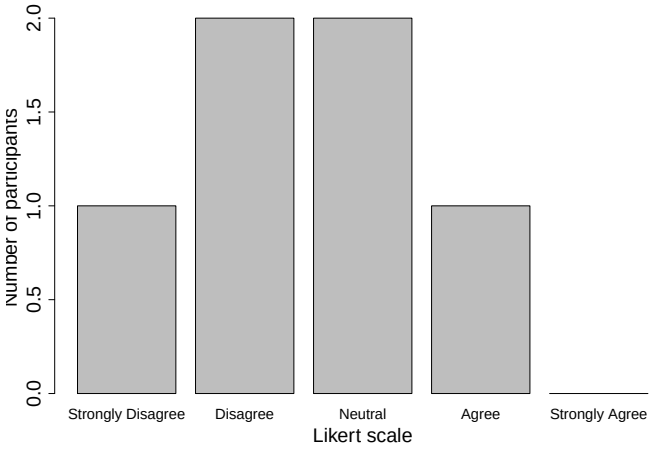

Figure 5.15.: Results for: I needed to learn a lot of things before I could get going with the system.

5.3.4. Package

The repository containing the materials for the understandability user study, results, and analysis scripts is accessible in [Hak24].

5.3.5. Validity Threats

There are several potential threats to the validity of this user study that must be discussed.

1. **Selection bias.** Whether or not the selected participants of the study represent the target population is a major threat. It makes sense for the code inspection to be performed by experienced developers. Experience refers to a prior practice in code reviewing and static analysis. Selecting only experienced developers cannot be representative either. Because not all developers integrating third-party code into their applications

have prior knowledge of being in a code reviewing team. We tried to select a mix of both groups to mitigate the threat. However, finding a balance was very difficult as students often focused on developing rather than doing code reviews. There is still a threat that a balance of both groups may yield a different result.

2. **Measurement bias.** Another major threat is measuring a subjective, biased, or insufficient comprehensive understandability score. The subjectivity and bias come from correcting the answers based on the subjective opinion of this dissertation's author. Although, we carefully designed a grading scheme to make the correction as systematic as possible. However, we can not entirely rule out the threat. It would have been better to ask a second or third person for correction. However, the subjectivity threat is still there. Concerning not being sufficiently comprehensive, there is a threat that only a quiz set is not comprehensive to assess understandability. Firstly, a team often inspects code. Secondly, there are various metrics for the complexity of the code, such as cyclomatic complexity, that could be combined with the quiz score. Thirdly, answering quizzes could be augmented with additional questionnaires to assess the understandability further. Concerning the choice of using one participant for code inspection, we chose implementations with relatively small lines of code that are doable by one person in a reasonable duration. Concerning various metrics and Questionnaires, another study should consider both.

3. **Learning effect.** When the same participant uses Imune and then code inspection or vice versa, there is a threat that the participant already reached a level of understanding using either method first. Moreover, for each method, there are two tasks, and going from one to the second has the threat that participants gain experience with the format of questions and hence less cognitive load. To mitigate this, we designed the user study to be a between-subject study where each participant only worked with one method. Moreover, we randomly assigned a different order for a participant, i.e., some worked on the

first and second while others worked on the second and then solved the first task.

4. **Sample Size** With a statistical power of 0.8, often the choice for user studies and an expected low effect size, a sample size of 200 is needed, while the sample size for an expected effect size of 0.5 is 60. The sample size in our evaluation is a major threat, and we can not entirely rule out the threat.

5. **Tasks** The level of task Complexity and prior knowledge issues associated with the design specification and codebase are two threats. Task complexity refers to the quizzes as well as the codebase. When the task is very complex, the threat is that probably no one can solve the task. On the other hand, too simple may produce an understandability score that does not indicate any differences between the two methods. To counteract, we provided two tasks subjectively thinking that are of different complexity yet not too simple or too complex. Furthermore, to ensure participants could answer questions, we made pilot runs and revised the questions and task descriptions accordingly. There is still a threat that participants may have prior knowledge of known issues of task implementation artifacts. Only two of the participants reported that they knew about the issue beforehand. Unfortunately, we did not systematically collect whether or not participants have prior knowledge of the known issues and familiarity with the codebase.

CHAPTER 6

Conclusion

This chapter summarizes the contributions and discusses open questions and possible directions for future works.

6.1. Summary

This dissertation presents two contributions to the behavior conformance checking process. First is the Imune method, specifically designed to support conformance checking of third-party components by application designers. Imune provides the state variable mapping model and explains the system behavior, including as-Intended behavior and conformance checking results. Second is understandability assessment using a comparative analysis which provides a basis for measuring the effectiveness of a behavioral conformance checking method concerning understandability.

Imune Method. Imune is a behavior conformance checking method for third-party components. In addition to formal method experts and library designers, Imune, in particular, recognizes the role of application designers

who are end-users of third-party components and not part of their design specification and implementation. Imune introduces a state variable mapping model and explanation of specification and conformance checking results to support the separation of concerns between the roles. State variable mapping models use the refinement mapping function that relates variables from implementation to those specified in the specification. The refinement mapping function in the Imune is for variables that take a string or integer values, a set of values, a sequence of values, function values, and auxiliary history variables. Concerning the explanation of as-Intended Behavior, Imune summarizes (1) all variables specified in a specification and domain of the values each variable takes, and (2) step-wise change in values of one or more variables. Conformance checking result is in the form of a graph, and each node (a.k.a. states) can be either (1) conforming, (2) diverged, (3) missing, or (4) new. The meaning of each node is the following.

- **Conforms**: The next observed state is marked as conforms in two cases. First, the next observed state is the first conforming state in the generated conformance graph, which *only* requires the same set of variables with equal values to a state in the design specification. Second, when not first anymore, the next observed state must relate to the next step transition in the design specification, which means the observed state conforms if and only if the *next step* transition in the design specification arrives at a state with equal variables value to the observed state.
- **Diverges**: When the next observed state is not a next-step transition according to the design specification but reachable with more than two steps, the observed state is marked as diverged, indicating that one or more states are expected to be seen from the system runtime behavior but is *missing*.
- **Missing**: Represents states that are not observed from the system runtime but are expected according to the next-step transition in the design specification.

- **New**: Indicates two cases. First, the observed state is unreachable from previous conforming states, including green or yellow-marked states. Second, the observed state contains variables with values not equal to one in the design specification.

Imune (1) guides formal method experts and third-party components designers, and (2) supports application designers. Concerning the first, formal method experts generate the reachability graph using the model-checking tool, and provide the configuration models for both specification variables and the state mapping functions. Modeling state mapping functions requires knowledge of system implementation. Hence, a collaboration between formal method experts and third-party development teams is often needed to understand how specification variables are implemented. After selecting the most important variables, formal method experts and third-party components designers should collaborate on how each variable from the specification is implemented at the level of source code or existing observable. If needed, a monitoring agent should be developed to observe fields and returned values from methods in classes related to the variables.

Validation. This dissertation evaluates Imune's effectiveness in (1) correctly identifying deviations and conformance and (2) understandability of application designers when using the approach. For the evaluation, we selected ReadWrite lock and Producer-Consumer design patterns specified and implemented by third parties. Producer-Consumer implementation incorrectly implements its specification, resulting in the deadlock at runtime. On the other hand, the ReadWrite lock correctly implements its inadequate specification that allows a task to hold either read or write while another task holds an exclusive WriteRead lock. Concerning correctness, Imune correctly identifies conformance and divergence between as-implemented and as-Intended behavior. We conducted a comparative analysis against code inspection through a user study with 10 participants, solving a set of understandability questions concerning ReadWrite lock and Proucer-Consumer projects.

The results show that, on average, participants scored 50.18% better when using Imune for understanding complex third-party Producer-Consumer components, including its as-Intended and as-implemented behaviors.

6.2. Limitations

Section 4.5 provides a list of assumptions and limitations concerning each contribution of this dissertation. Hence, this section only references the respective section.

6.3. Future Work

This section provides an overview of potential future works and remaining open questions.

- **Automating State Variable Mapping.** State variable mapping process involves creating refinement mapping functions that map variables in the implementation to variables in the specification. Imune provides a configuration model for describing the state mapping function. However, the entire process is manual. It is interesting to investigate how much the process of identifying the mapping functions can be automated. Artificial Intelligence (AI) can play a significant role. Here are some thoughts.

  Concerning specification variables, AI techniques, such as NLP, can be used to automatically extract variables and their comments from natural language specification documents, which helps in identifying relevant variables and their meanings, reducing the manual effort required for variable identification. AI-powered program analysis techniques can be employed to analyze both the implementation and extract variables related to the specification. The analysis can be static, dynamic, or a combination of both. Concerning mapping functions, AI algorithms can be trained on a dataset of manually mapped variables

to learn patterns and relationships between variables in the specification and implementation of various systems. These models can then automatically infer variable mappings, reducing the need to create mapping functions manually.

- **Validation on Real-world Case Study.** This dissertation does not validate the use of Imune by conducting a case study. Here are our thoughts about settings for such a case study. The software system requires integrating a third-party component, such as a key-value store database or a payment gateway, into the existing architecture. The formal method expert role creates design specifications of the third-party components and creates required artifacts for Imune in collaboration with third-party component designers. Application designers must integrate Imune into their testing or operation phases to check the third-party component's behavioral conformance.

  The evaluation should consider effectiveness in terms of (1) correctness, (2) the required effort by developers or operators in learning and using Imune, (3) tooling issues, (4) whether or not existing structure for variable mapping provided enough coverage for the mapping scenarios related to the third-party component and (5) whether or not application designers are confident that they understand the behavior of the third-party.

- **Extension to Dynamic and Evolving Systems.** Nowadays, the frequent release of software systems is common. The state variable mapping model likely requires adjustments when implementation-level variables change. The change can be a variable name or underlying semantics. Possible future work is to investigate techniques to capture and represent changes in system behavior over time and adapt the state variable mapping model and explanation generation process accordingly.

- **Enhancing Explanation Generation.** Explore techniques to enhance the generation of explanations for the as-intended behavior and con-

formance results. The explanation could involve developing more sophisticated algorithms or leveraging natural language processing techniques to generate more comprehensive and user-friendly explanations.

Bibliography

[Aba15] M. Abadi. 'The Prophecy of Undo'. In: *Fundamental Approaches to Software Engineering - 18th International Conference, FASE 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. Ed. by A. Egyed, I. Schaefer. Vol. 9033. Lecture Notes in Computer Science. Springer, 2015, pp. 347–361. URL: https://doi.org/10.1007/978-3-662-46675-9%5C_23 (cit. on p. 97).

[ABL02] G. Ammons, R. Bodıék, J. R. Larus. 'Mining specifications'. In: *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*. Ed. by J. Launchbury, J. C. Mitchell. ACM, 2002, pp. 4–16. URL: https://doi.org/10.1145/503272.503275 (cit. on p. 46).

[Abr10] J. Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010. URL: https://doi.org/10.1017/CBO9781139195881 (cit. on p. 29).

[AGR11] P. Arcaini, A. Gargantini, E. Riccobene. 'CoMA: Conformance Monitoring of Java Programs by Abstract State Machines'. In: *Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers*. Ed. by S. Khurshid, K. Sen. Vol. 7186. Lecture Notes in Computer Science. Springer, 2011, pp. 223–238. URL: https://doi.org/10.1007/978-3-642-29860-8_17 (cit. on pp. 3, 50, 54, 56, 59).

[aio22] aiorwlock contributors. *Code repository*. https://github.com/aio-libs/aiorwlock/tree/v0.4.0. Accessed: 2023-July. 2022 (cit. on p. 101).

[ALC09] C. Ackermann, M. Lindvall, R. Cleaveland. 'Towards Behavioral Reflexion Models'. In: *ISSRE 2009, 20th International Symposium on Software Reliability Engineering, Mysuru, Karnataka, India, 16-19 November 2009*. IEEE Computer Society, 2009, pp. 175–184. URL: https://doi.org/10.1109/ISSRE.2009.27 (cit. on pp. 49, 54, 56, 59).

[Ale21] Alexander Niederbühl. *Code repository*. https://github.com/Alexander-N/tla-specs/tree/main/aiorwlock. Accessed: 2023-July. 2021 (cit. on p. 101).

[ALRL04] A. Avizienis, J. Laprie, B. Randell, C. E. Landwehr. 'Basic Concepts and Taxonomy of Dependable and Secure Computing'. In: *IEEE Trans. Dependable Secur. Comput.* 1.1 (2004), pp. 11–33. URL: https://doi.org/10.1109/TDSC.2004.2 (cit. on pp. 99, 101).

[BAE+15] J. Buckley, N. Ali, M. English, J. Rosik, S. Herold. 'Real-Time Reflexion Modelling in architecture reconciliation: A multi case study'. In: *Inf. Softw. Technol.* 61 (2015), pp. 107–123. URL: https://doi.org/10.1016/j.infsof.2015.01.011 (cit. on p. 42).

[BCC+05] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, E. Poll. 'An overview of JML tools and applications'. In: *Int. J. Softw. Tools Technol. Transf.* 7.3 (2005), pp. 212–232. URL: https://doi.org/10.1007/s10009-004-0167-4 (cit. on pp. 20, 46, 53).

[BCK12] L. J. Bass, P. C. Clements, R. Kazman. *Software architecture in practice*. SEI series in software engineering. Addison-Wesley-Professional, 2012 (cit. on p. 36).

[BGMT06] G. Bolch, S. Greiner, H. de Meer, K. S. Trivedi. *Queueing Networks and Markov Chains - Modeling and Performance Evaluation with Computer Science Applications, Second Edition*. Wiley, 2006. URL: http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0471565253.html (cit. on p. 3).

[BHJM07] D. Beyer, T. A. Henzinger, R. Jhala, R. Majumdar. 'The software model checker Blast'. In: *Int. J. Softw. Tools Technol. Transf.* 9.5-6 (2007), pp. 505–525. URL: https://doi.org/10.1007/s10009-007-0044-z (cit. on p. 20).

[BK08] C. Baier, J. Katoen. *Principles of model checking*. MIT Press, 2008 (cit. on pp. 4, 22, 24, 26–29, 33, 38).

[Bro+96] J. Brooke et al. 'SUS-A quick and dirty usability scale'. In: *Usability evaluation in industry* 189.194 (1996), pp. 4–7 (cit. on p. 119).

[BS03] E. Börger, R. F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003. URL: http://www.springer.com/computer/swe/book/978-3-540-00702-9 (cit. on p. 50).

[CdR05] F. Chen, M. d'Amorim, G. Rosu. 'Checking and Correcting Behaviors of Java Programs at Runtime with Java-MOP'. In: *Proceedings of the Fifth Workshop on Runtime Verification, RV@CAV 2005, Edinburgh, UK, July 12, 2005*. Ed. by H. Barringer, B. Finkbeiner, Y. Gurevich, H. Sipma. Vol. 144. Electronic Notes in Theoretical Computer Science 4. Elsevier, 2005, pp. 3–20. URL: https://doi.org/10.1016/j.entcs.2006.02.002 (cit. on p. 57).

[CFP96] A. R. Cavalli, J. P. Favreau, M. Phalippou. 'Standardization of Formal Methods in Conformance Testing of Communication Protocols'. In: *Comput. Networks ISDN Syst.* 29.1 (1996), pp. 3–14. URL: https://doi.org/10.1016/S0169-7552(96)00015-3 (cit. on p. 41).

[CHV18] E. M. Clarke, T. A. Henzinger, H. Veith. 'Introduction to Model Checking'. In: *Handbook of Model Checking*. Ed. by E. M. Clarke, T. A. Henzinger, H. Veith, R. Bloem. Springer, 2018, pp. 1–26. URL: https://doi.org/10.1007/978-3-319-10575-8%5C_1 (cit. on pp. 21, 29, 45, 47).

[CI90] E. J. Chikofsky, J. H. C. II. 'Reverse Engineering and Design Recovery: A Taxonomy'. In: *IEEE Softw.* 7.1 (1990), pp. 13–17. URL: https://doi.org/10.1109/52.43044 (cit. on p. 36).

[CPC11] G. Canfora, M. D. Penta, L. Cerulo. 'Achievements and challenges in software reverse engineering'. In: *Commun. ACM* 54.4 (2011), pp. 142–151. URL: https://doi.org/10.1145/1924421.1924451 (cit. on pp. 16, 36).

[CR05] F. Chen, G. Rosu. 'Java-MOP: A Monitoring Oriented Programming Environment for Java'. In: *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*. Ed. by N. Halbwachs, L. D. Zuck. Vol. 3440. Lecture Notes in Computer Science. Springer, 2005, pp. 546–550. URL: https://doi.org/10.1007/978-3-540-31980-1%5C_36 (cit. on p. 57).

[CW96] E. M. Clarke, J. M. Wing. 'Formal Methods: State of the Art and Future Directions'. In: *ACM Comput. Surv.* 28.4 (1996), pp. 626–643. URL: https://doi.org/10.1145/242223.242257 (cit. on p. 20).

[Dav19] C. Davis. *Cloud Native Patterns: Designing Change-tolerant Software*. Manning Publications, 2019 (cit. on pp. 1, 2).

[DSMB20] S. Dunzer, M. Stierle, M. Matzner, S. Baier. 'Conformance checking: A state-of-the-art literature review'. In: *CoRR* abs/2007.10903 (2020). URL: https://arxiv.org/abs/2007.10903 (cit. on pp. 49, 54).

[EPG+07] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, C. Xiao. 'The Daikon system for dynamic detection of likely invariants'. In: *Sci. Comput. Program.* 69.1-3 (2007), pp. 35–45. URL: https://doi.org/10.1016/j.scico.2007.01.015 (cit. on pp. 3, 46).

[Fag02] M. E. Fagan. 'A History of Software Inspections'. In: *Software Pioneers*. Ed. by M. Broy, E. Denert. Springer Berlin Heidelberg, 2002, pp. 562–573. URL: https://doi.org/10.1007/978-3-642-59412-0%5C_34 (cit. on p. 9).

[Fag76] M. E. Fagan. 'Design and Code Inspections to Reduce Errors in Program Development'. In: *IBM Syst. J.* 15.3 (1976), pp. 182–211. URL: https://doi.org/10.1147/sj.153.0182 (cit. on p. 9).

[FHR13] Y. Falcone, K. Havelund, G. Reger. 'A Tutorial on Runtime Verification'. In: *Engineering Dependable Software Systems*. Ed. by M. Broy, D. A. Peled, G. Kalus. Vol. 34. NATO Science for Peace and Security Series, D: Information and Communication Security. IOS Press, 2013, pp. 141–175. URL: https://doi.org/10.3233/978-1-61499-207-3-141 (cit. on pp. 3, 46, 47).

[FKRT21] Y. Falcone, S. Krstic, G. Reger, D. Traytel. 'A taxonomy for classifying runtime verification tools'. In: *Int. J. Softw. Tools Technol. Transf.* 23.2 (2021), pp. 255–284. URL: https://doi.org/10.1007/s10009-021-00609-z (cit. on p. 46).

[Hak20] M. A. Hakamian. 'Engineering Resilience: Predicting The Change Impact on Performance and Availability of Reconfigurable Systems'. In: *2020 IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops, Coimbra, Portugal, October 12-15, 2020*. IEEE, 2020, pp. 143–146. URL: https://doi.org/10.1109/ISSREW51248.2020.00054 (cit. on p. 65).

[Hak24] A. Hakamian. *supplementary materials*. Apr. 2024. URL: https://doi.org/10.5281/zenodo.11092902 (cit. on pp. 11, 138).

[HKF+22] M. A. Hakamian, F. Klinaku, S. Frank, A. van Hoorn, S. Becker. 'Planning for Software System Recovery by Knowing Design Limitations of Cloud-native Patterns'. In: *Proceedings of the 12th International Conference on Cloud Computing and Services Science, CLOSER 2022,, Online Streaming, April 27-29, 2022*. Ed. by M. van Steen, D. Ferguson, C. Pahl. SCITEPRESS, 2022, pp. 215–222. URL: https://doi.org/10.5220/0011061500003200 (cit. on pp. 66, 67, 75).

[HKHB20] M. A. Hakamian, F. Klinaku, A. van Hoorn, S. Becker. 'Resilience, Survivability, and Elasticity: A Taxonomy for Change Impact Quantification of Reconfigurable Systems'. In: *2020 IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops, Coimbra, Portugal, October 12-15, 2020*. IEEE, 2020, pp. 267–274. URL: https://doi.org/10.1109/ISSREW51248.2020.00084 (cit. on p. 64).

[HMPR04] A. R. Hevner, S. T. March, J. Park, S. Ram. 'Design Science in Information Systems Research'. In: *Management Information Systems Quarterly*. 28.1 (2004), pp. 75–105. URL: http://misq.org/design-science-in-information-systems-research.html (cit. on p. 10).

[HP00] K. Havelund, T. Pressburger. 'Model Checking JAVA Programs using JAVA PathFinder'. In: *Int. J. Softw. Tools Technol. Transf.* 2.4 (2000), pp. 366–381. URL: https://doi.org/10.1007/s100090050043 (cit. on pp. 20, 47, 53, 57).

[HR01] K. Havelund, G. Rosu. 'Monitoring Java Programs with Java PathExplorer'. In: *Workshop on Runtime Verification, RV 2001, in connection with CAV 2001, Paris, France, July 23, 2001*. Ed. by K. Havelund, G. Rosu. Vol. 55. Electronic Notes in Theoretical Computer Science 2. Elsevier, 2001, pp. 200–217. URL: https://doi.org/10.1016/S1571-0661(04)00253-1 (cit. on pp. 46, 53).

[Hys12] Hystrix Contributors. *Hystrix Documentation*. https://github.com/Netflix/Hystrix. Accessed: 2023-July. 2012 (cit. on p. 7).

[IH19] B. Ibryam, R. Huß. *Kubernetes patterns: reusable elements for designing cloud-native applications*. O'Reilly Media, 2019 (cit. on p. 1).

[ISO15] ISO/IEC/IEEE 15288. *Systems and software engineering, System life cycle processes*. 2015. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7106435 (cit. on pp. 7, 72).

[KC07] B. A. Kitchenham, S. Charters. *Guidelines for performing Systematic Literature Reviews in Software Engineering*. Tech. rep. Keele University and Durham University Joint Report, 2007. URL: https://www.elsevier.com/__data/promis_misc/525444systematicreviewsguide.pdf (cit. on p. 19).

[Koy90] R. Koymans. 'Specifying Real-Time Properties with Metric Temporal Logic'. In: *Real Time Syst.* 2.4 (1990), pp. 255–299. URL: https://doi.org/10.1007/BF01995674 (cit. on p. 45).

[Kru95] P. Kruchten. 'The 4+1 View Model of Architecture'. In: *IEEE Softw.* 12.6 (1995), pp. 42–50. URL: https://doi.org/10.1109/52.469759 (cit. on p. 36).

[KVB+99] M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, O. Sokolsky. 'Formally specified monitoring of temporal properties'. In: *11th Euromicro Conference on Real-Time Systems (ECRTS 1999), Proceedings*. IEEE Computer Society, 1999, pp. 114–122. URL: https://doi.org/10.1109/EMRTS.1999.777457 (cit. on p. 46).

[Lam02] L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002. URL: http://research.microsoft.com/users/lamport/tla/book.html (cit. on pp. 23, 34, 82).

[Lam94] L. Lamport. 'The Temporal Logic of Actions'. In: *ACM Trans. Program. Lang. Syst.* 16.3 (1994), pp. 872–923. URL: https://doi.org/10.1145/177492.177726 (cit. on pp. 21, 23).

[Lap08] J.-C. Laprie. 'From dependability to resilience'. In: *38th IEEE/IFIP Int. Conf. On Dependable Systems and Networks*. 2008, G8–G9 (cit. on p. 2).

[LM17] L. Lamport, S. Merz. 'Auxiliary Variables in TLA+'. In: *CoRR* abs/1703.05121 (2017). URL: http://arxiv.org/abs/1703.05121 (cit. on p. 84).

[LMP08] D. Lorenzoli, L. Mariani, M. Pezzè. 'Automatic generation of software behavioral models'. In: *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*. Ed. by W. Schäfer, M. B. Dwyer, V. Gruhn. ACM, 2008, pp. 501–510. URL: https://doi.org/10.1145/1368088.1368157 (cit. on p. 3).

[Mar22] Markus Kuppe. *Code repository*. https://github.com/lemmy/BlockingQueue. Accessed: 2023-July. 2022 (cit. on p. 101).

[Mey97] B. Meyer. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997. URL: http://www.eiffel.com/doc/oosc/page.html (cit. on p. 45).

[Mic21] Michel Charpentier. *Debugging Concurrent Systems*. https://levelup.gitconnected.com/debugging-concurrent-systems-with-a-model-checker-c7eee210d86f. Accessed: 2023-July. 2021 (cit. on p. 101).

[MJG+12] P. O. Meredith, D. Jin, D. Griffith, F. Chen, G. Rosu. 'An overview of the MOP runtime verification framework'. In: *Int. J. Softw. Tools Technol. Transf.* 14.3 (2012), pp. 249–289. URL: https://doi.org/10.1007/s10009-011-0198-6 (cit. on pp. 3, 20, 46, 53, 55, 57).

[MM89] E. W. Myers, W. Miller. 'Approximate matching of regular expressions'. In: *Bulletin of mathematical biology* 51.1 (1989), pp. 5–37 (cit. on p. 56).

[MN97] G. C. Murphy, D. Notkin. 'Reengineering with reflexion models: A case study'. In: *Computer* 30.8 (1997), pp. 29–36 (cit. on p. 58).

[MNS01] G. C. Murphy, D. Notkin, K. J. Sullivan. 'Software Reflexion Models: Bridging the Gap between Design and Implementation'. In: *IEEE Trans. Software Eng.* 27.4 (2001), pp. 364–380. URL: https://doi.org/10.1109/32.917525 (cit. on pp. 20, 48, 53, 56, 58).

[MTZ17] F. U. Muram, H. Tran, U. Zdun. 'Systematic Review of Software Behavioral Model Consistency Checking'. In: *ACM Comput. Surv.* 50.2 (2017), 17:1–17:39. URL: https://doi.org/10.1145/3037755 (cit. on p. 49).

[Net04] N. Nethercote. 'Dynamic binary analysis and instrumentation: or building tools is easy'. PhD thesis. University of Cambridge, UK, 2004. URL: https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.616254 (cit. on p. 40).

[New14] C. Newcombe. 'Why Amazon Chose TLA +'. In: *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 4th International Conference, ABZ 2014*. Springer, 2014, pp. 25–39. URL: https://doi.org/10.1007/978-3-662-43652-3%5C_3 (cit. on p. 66).

[Nic18a] A. Nicolaescu. 'Behavior-Based Architecture Conformance Checking'. PhD thesis. RWTH Aachen University, Germany, 2018. URL: http://publications.rwth-aachen.de/record/751838 (cit. on pp. 53, 56, 59).

[Nic18b] A. Nicolaescu. 'Behavior-Based Architecture Conformance Checking'. In: *Ausgezeichnete Informatikdissertationen 2018*. Vol. D-19. LNI. GI, 2018, pp. 161–170. URL: https://dl.gi.de/20.500.12116/36732 (cit. on p. 56).

[NL16] A. Nicolaescu, H. Lichter. 'Behavior-Based Architecture Reconstruction and Conformance Checking'. In: *13th Working IEEE/IFIP Conference on Software Architecture, WICSA 2016, Venice, Italy, April 5-8, 2016*. IEEE Computer Society, 2016, pp. 152–157. URL: https://doi.org/10.1109/WICSA.2016.25 (cit. on p. 56).

[NNH99] F. Nielson, H. R. Nielson, C. Hankin. *Principles of program analysis*. Springer, 1999. URL: https://doi.org/10.1007/978-3-662-03811-6 (cit. on p. 39).

[Nyg18] M. T. Nygard. *Release it!: design and deploy production-ready software*. Pragmatic Bookshelf, 2018 (cit. on pp. 2, 12, 75).

[OO14] D. Ongaro, J. K. Ousterhout. 'In Search of an Understandable Consensus Algorithm'. In: *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*. Ed. by G. Gibson, N. Zeldovich. USENIX Association, 2014, pp. 305–319. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro (cit. on p. 119).

[Ora23] Oracle. *Java 2 Platform, Enterprise Edition (J2EE) Overview*. 2023. URL: https://www.oracle.com/java/technologies/appmodel.html (cit. on p. 1).

[Poh10] K. Pohl. *Requirements Engineering - Fundamentals, Principles, and Techniques*. Springer, 2010 (cit. on p. 21).

[Pos03] A. Postma. 'A method for module architecture verification and its application on a large component-based system'. In: *Inf. Softw. Technol.* 45.4 (2003), pp. 171–194. URL: https://doi.org/10.1016/S0950-5849(02)00193-3 (cit. on pp. 48, 53, 58).

[PTV+10] L. T. Passos, R. Terra, M. T. Valente, R. Diniz, N. C. Mendonça. 'Static Architecture-Conformance Checking: An Illustrative Overview'. In: *IEEE Softw.* 27.5 (2010), pp. 82–89. URL: https://doi.org/10.1109/MS.2009.117 (cit. on p. 48).

[SAG+06] B. R. Schmerl, J. Aldrich, D. Garlan, R. Kazman, H. Yan. 'Discovering Architectures from Running Systems'. In: *IEEE Trans. Software Eng.* 32.7 (2006), pp. 454–466. URL: https://doi.org/10.1109/TSE.2006.66 (cit. on p. 3).

[SB13] L. de Silva, D. Balasubramaniam. 'PANDArch: A Pluggable Automated Non-intrusive Dynamic Architecture Conformance Checker'. In: *Software Architecture - 7th European Conference, ECSA 2013, Montpellier, France, July 1-5, 2013. Proceedings*. Vol. 7957. Lecture Notes in Computer Science. Springer, 2013, pp. 240–248. URL: https://doi.org/10.1007/978-3-642-39031-9%5C_21 (cit. on pp. 42, 48).

[SDH20] J. Schvimer, A. J. J. Davis, M. Hirschhorn. 'eXtreme Modelling in Practice'. In: *Proc. VLDB Endow.* 13.9 (2020), pp. 1346–1358. URL: http://www.vldb.org/pvldb/vol13/p1346-schvimer.pdf (cit. on pp. 5, 51, 54, 56, 59, 66).

[SSAE19] C. Sánchez, G. Schneider, W. Ahrendt, e. a. Ezio Bartocci. 'A survey of challenges for runtime verification from advanced application domains (beyond software)'. In: *Formal Methods Syst. Des.* 54.3 (2019), pp. 279–335. URL: https://doi.org/10.1007/s10703-019-00337-w (cit. on p. 46).

[TB17] K. S. Trivedi, A. Bobbio. *Reliability and Availability Engineering - Modeling, Analysis, and Applications*. Cambridge University Press, 2017. URL: https://www.cambridge.org/de/academic/subjects/engineering/engineering-general-interest/reliability-and-availability-engineering-modeling-analysis-and-applications (cit. on pp. 3, 20).

[TCL04] R. T. Tvedt, P. Costa, M. Lindvall. 'Evaluating software architectures'. In: *Adv. Comput.* 61 (2004), pp. 1–43. URL: https://doi.org/10.1016/S0065-2458(03)61001-6 (cit. on pp. 8, 43, 44).

[WA15] C. Wohlin, A. Aurum. 'Towards a decision-making structure for selecting a research design in empirical software engineering'. In: *Empir. Softw. Eng.* 20.6 (2015), pp. 1427–1455. URL: https://doi.org/10.1007/s10664-014-9319-7 (cit. on pp. 10, 63, 64).

[WRH+12] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell. *Experimentation in Software Engineering*. Springer, 2012. URL: https://doi.org/10.1007/978-3-642-29044-2 (cit. on pp. 11, 62, 116, 118, 125).

Appendix

This chapter presents the materials regarding the user study for understandability assessment.

A.1. Producer-Consumer Experiment

This section provides questions with respect to both Producer-Consumer and ReaWrite lock projects. The provided questions are from code inspection and if a question for Imune is slightly different, the difference is mentioned.

Here are the questions and the correction scheme regarding as-Intended behavior. Note that each question is worth 3 points.

1. In what condition do producers and consumers have to wait?

   Answer. Producers have to wait, if the queue is full (BlockingQueue.java lines 18f.) and Coonsumers have to wait, if the queue is empty (BlockingQueue.java, lines 36f.)

2. Is it intended that the system execution eventually goes into a state where progress is not possible anymore? i.e., all producers and consumers wait

   Answer. No, because it means deadlock and that is not intended (pattern description, README.md line 6)

Following are questions with respect to as-implemented behavior.

1. In what condition do producers and consumers have to wait?

   Answer. Producers have to wait if the queue is full (BlockingQueue line 24/26) and Consumers have to wait if the queue is empty (BlockingQueue line 42/44).

2. What happens to the set of waiting tasks after a producer produces data into the buffer?

   Answer. An arbitrary task is notified (BlockingQueue line 29).

3. What happens to the set of waiting tasks after consumers consume data from buffer?

   Answer. An arbitrary task is notified (BlockingQueue line 29).

4. Based on your understanding from all observed behaviors, do you predict that the system execution eventually goes into a state where progress is not possible anymore? i.e., all producers and consumers wait

   Answer. Yes, it is possible. Because for example after producing a data, the buffer gets full and the producer may notifies another producer instead of a consumer.

A.2. ReadWrite Lock Experiment

Concerning as-designed behavior, here are the questions, and the correction scheme.

1. What happens to the state of the shared resource when a task acquires read lock? Note that write to WriteRead shows the task acquired the read lock in addition to the write lock.

   Answer. What happens to the state is not precisely defined. If (shared) read access is obtained, the state should be positive afterwards (according to the pattern description). If read lock is acquired in addition to write lock, there is no shared read access. Therefore, the number must not be positive, and also not 0, as the resource is not unlocked. However, the pattern description is not precise about WriteRead state.

2. What happens to the state of the shared resource when a task acquires a write lock?

   Answer. The state should be changed to a negative number, according to the pattern description.

3. What happens to the state of the shared resource when a task releases a read lock? Note that WriteRead to write shows the task only released the read lock.

   Answer. Line 53 hints that the value should be decreased by 1 and gets closer to 0. However, the pattern description is not precise.

4. What happens to the state of the shared resource when a task releases a write lock? Note that WriteRead to read shows the task only released the write lock.

   Answer. Line 53 hints that as the state should be increased by 1 and gets closer to 0. However, the pattern description is not precise.

5. Are there any states where a task acquired either a read lock or a write lock while another task acquired a WriteRead lock on the shared resource?

   Answer. No, WriteRead lock is exclusive, according to the pattern description.

Following are questions with respect to as-implemented behavior.

1. What happens to the state of the shared resource when a task acquires read lock? Note that write to WriteRead shows the task acquired the read lock in addition to the write lock.

   Answer. Looking at the lines 70, 77, or 87, The state is increased by one. This also happens if the task already has the write lock.

2. What happens to the state of the shared resource when a task acquires a write lock?

   Answer. Looking at the lines the state is decreased by one

3. What happens to the state of the shared resource when a task releases a read lock? Note that WriteRead to write shows the task only released the read lock.

   Answer. What happens only depends on the value of state, not the actual state of the lock. If the value is > 0, it is decreased by 1, otherwise increased by 1 (lines 131-134).

4. What happens to the state of the shared resource when a task releases a write lock? Note that WriteRead to read shows the task only released the write lock.

   Answer. What happens only depends on the value of state, not the actual state of the lock. If the value is > 0, it is decreased by 1, otherwise increased by 1 (lines 131-134).

5. Based on your understanding from all observed behaviors, do you predict that the system execution will eventually reach a state where a task acquires either a read lock or a write lock while another task already has a WriteRead lock on the shared resource?

   Answer. To get a WriteRead, the state is first decremented (by getting write), then incremeted again (by getting read), resulting in a state of 0. Thus, it would be interpreted as unlocked, allowing others to gain read/write access.