Universität Stuttgart

Institute for Formal Methods of Computer Science

Universitätsstraße 38
70569 Stuttgart

Masterarbeit

Analysing Timing Behavior of Component-Based Software Systems

Aaron Hilbig

Studiengang:	Informatik
1. Prüfer:	PD Dr. Manfred Kufleitner
2. Prüfer:	
Betreuer:	Dennis Hendriks, Dr. Wytse Oortwijn
begonnen am:	20.08.2022
beendet am:	20.02.2023

Institute for Formal Methods of Computer Science
University of Stuttgart
Universitätsstraße 38, 70569 Stuttgart

Master Thesis

Analysing Timing Behavior of Component-Based Software Systems

Aaron Hilbig

Course of study:	Informatik
Examiner:	PD Dr. Manfred Kufleitner
Supervisors:	Dennis Hendriks
	Dr. Wytse Oortwijn
Started:	August 20, 2022
Completed:	February 20, 2023

Abstract

Component-based Software Engineering (CBSE) is an established approach for dealing with the complexity of large-scale software systems. In such systems, the impact of small software changes on the overall system can be hard to assess. Because the behavior of the fully integrated system can be difficult to reason about for developers who know only small parts in-depth, the system can break in unexpected ways. This can also make developers increasingly reluctant to make any changes to the codebase for fear of regressions, leading to a degradation of code quality and maintainability.

CBSE is applied in a wide range of application domains in the industry. At ASML, it is used in the controller software of lithography machines. These complex cyber-physical systems consisting of hundreds of components are at the core of modern semiconductor manufacturing. If such a machine is defective in production, it can cost the operator "thousands of euros per minute"[1] until the issue is resolved. The motivation to avoid software bugs, regressions in performance and throughput losses is thus high.

To this end, researchers at ASML and TNO have developed the analysis tool *MIDS*. MIDS infers a sound approximation of the behavior of the overall system from the output traces of a machines' components. Using MIDS, models learned from the output traces of different software versions can be compared and the changes visualized. This provides a concise overview of the system and of detected behavioral changes between versions to the developers, enabling them to find unexpected changes and regressions. The methodology is not limited to ASML software, but is in principle generally applicable to component-based software. However, MIDS only supports the analysis of functional software changes, i.e., added, removed or moved behavior. Finding and analysing the root cause of performance issues in the systems is still an arduous manual process.

In this work, we aim to help engineers in finding performance issues or timing behavior changes in component-based software systems by leveraging the automata-based system models inferred by MIDS and adding timing information to them. We explore how this data can be extracted from the output traces, added to the system models, and visualized. We further investigate whether significant timing behavior changes can be automatically detected, to help engineers focus only on the parts of the system which are impacted by a timing change. We implement our approach as an extension to MIDS and evaluate it with artificially modified traces as well as with real-world traces. In our case studies, our prototype was not only able to detect the expected timing changes, but also detected additional timing changes that were unknown to the engineers. We received very positive feedback from the ASML engineers, who approached us independently for additional analyses and showed great interest in integrating our prototype into the main-line version of MIDS.

In a proof-of-concept with the open-source software cURL, we show that our methodology may even be applicable to non-component-based software and for analysing timing changes not caused by software changes.

[1] https://www.asml.com/en/products/customer-support

Zusammenfassung

Komponentenbasierte Softwareentwicklung (CBSE) ist ein etablierter Ansatz für den Umgang mit Komplexität in großen Softwaresystemen. In solchen Systemen können die Auswirkungen von Softwareänderungen auf das Gesamtsystem schwer abzuschätzen sein. Da das vollständig integrierte System ist für Entwickler, die nur kleine Teile des Systems im Detail kennen, nur schwer zu überschauen ist, können schon kleine Änderungen Probleme an unbekannten oder unerwarteten Stellen verursachen. Dies kann auch dazu führen, dass Entwickler aus Angst vor Regressionen zunehmend ungern Änderungen an der Codebasis vornehmen, was zu einer Verschlechterung der Codequalität und der Wartbarkeit führt.

CBSE wird von der Industrie in vielen unterschiedlichen Anwendungsbereichen genutzt. Bei ASML wird es bei der Entwicklung der Steuerungssoftware von Lithographiemaschinen eingesetzt. Diese komplexen cyber-physischen Systeme mit hunderten Komponenten sind das Herzstück der modernen Halbleiterfertigung. Fällt ein solches System in der Produktion aus, kann das den Betreiber "mehrere tausend Euro pro Minute"[2] kosten, bis das Problem behoben ist. Dementsprechend hoch ist die Motivation, Softwarebugs, Timing-Probleme und Durchsatzverluste zu vermeiden.

Zu diesem Zweck hat ASML in Zusammenarbeit mit TNO das Analysetool *MIDS* entwickelt. MIDS rekonstruiert aus den textbasierten Logs der Komponenten einer Maschine ein Verhaltensmodell des Gesamtsystems. Modelle, die aus den Logs verschiedener Softwareversionen gelernt wurden, können mit MIDS verglichen und deren Änderungen visualisiert werden. So wird den Entwicklern ein konziser Überblick über das System und über erkannte Verhaltensänderungen zwischen den Versionen zugänglich gemacht und ermöglicht es ihnen, unerwartete Änderungen und Regressionen zu finden. Die angewandten Methoden sind nicht auf ASML-Software beschränkt, sondern im Prinzip mit beliebiger komponentenbasierter Software nutzbar. MIDS unterstützt allerdings bisher nur die Analyse funktionaler Softwareänderungen, d.h. hinzugefügter, entfernter oder verschobener Funktionalitäten. Das Suchen und Analysieren von Performance-Problemen ist nach wie vor ein mühsamer manueller Prozess.

Ziel dieser Arbeit ist, Entwicklern das Untersuchen von Durchsatzverlusten und Timing-Änderungen in komponentenbasierter Software zu erleichtern, indem die von MIDS erlernten Systemmodelle um Zeitinformationen erweitert werden. Wir untersuchen, wie Zeitdaten aus den Logs extrahiert, zu den Systemmodellen hinzugefügt und visualisiert werden können. Weiterhin untersuchen wir, inwieweit Änderungen in zeitlichen Abläufen automatisiert erkannt werden können, um Entwicklern die Arbeit weiter zu erleichtern. Wir implementieren unseren Ansatz als Erweiterung zu MIDS und evaluieren ihn sowohl auf künstlich modifizierten Logs, als auch auf Logs aus realen Systemen. In unseren Fallstudien war unser Prototyp nicht nur in der Lage, die erwarteten Timing-Änderungen zu erkennen, sondern konnte auch zusätzliche Timing-Änderungen aufdecken, die den

[2] https://www.asml.com/en/products/customer-support

Entwicklern unbekannt waren. Wir erhielten positive Rückmeldungen seitens der Entwickler bei ASML, die im Laufe des Projekts unabh ngig weitere Analysen anforderten und Interesse an einer Integration unseres Prototyps in die Hauptversion von Mids zeigten.

Anhand eines Proof-of-Concept mit der Open Source Software cURL zeigen wir, dass unsere Methode möglicherweise sogar bei nicht-komponentenbasierter Software und für die Analyse von nicht durch Softwareänderungen hervorgerufenen Timing-Änderungen einsetzbar ist.

Contents

1 Introduction

Component-based Software Engineering (CBSE) is a well-established and thoroughly studied approach to manage the complexity of large-scale software systems [21, 27, 39]. CBSE is applied in a wide range of domains, from cloud computing and avionics to nuclear systems, to increase developer productivity, software quality, and maintainability [39]. To achieve this, a software system is divided into reusable *components* that communicate via loosely coupled *interfaces* [27, 39]. Each component can be developed, tested and maintained in isolation [16, 21].

At ASML, CBSE is used for the controller software of lithography machines for semiconductor manufacturing. Figure 1.1 shows one such machine. These complex cyber-physical systems contain hundreds of independent communicating components.

The large production volumes in semiconductor manufacturing and ever-increasing demand necessitate smooth operations at high throughput. Speed is a priority – a millisecond change in one component may already impact the customer's turnover. If a system is defective, it can cost the operator ”thousands of euros per minute”[1] until the issue is resolved. The motivation to avoid both software bugs and regressions in performance in the controller software is thus high.

CBSE can also add new challenges to the engineering process [10]. The impact of small software changes on the complete system can be hard to assess, and the system may break in unexpected ways, in particular if systems are too large for a single developer to reason about or include legacy components. Developers may even become unwilling to make any changes to the codebase in fear of regressions, leading to a degradation of code quality and maintainability.

[1] Source: https://www.asml.com/en/products/customer-support, Accessed 2022-09-07.

Figure 1.1: ASML TWINSCAN NXE:3400 system, with closed and open front. Source: ASML Media Library, https://asml.picturepark.com/s/AB09FG2k, Accessed 2022-09-07. Copyright ©ASML.

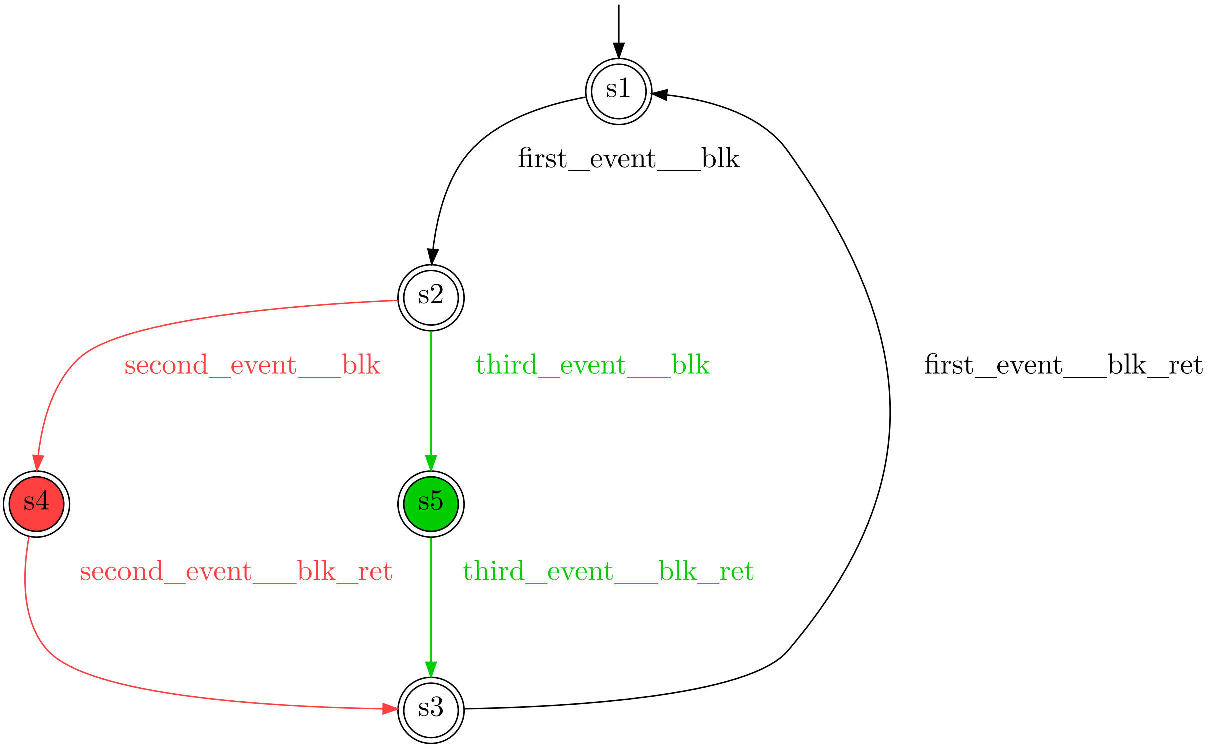


Figure 1.2: System behavior model diagram as presented by Mids. Red/green colors indicate functionality that has been removed/added between software versions.

1.1 Learning and Comparing System Behavior Models

To help with the challenges of CBSE and in maintaining the highly complex controller software, researchers at TNO-ESI and ASML have developed a tool called Model Inference and Development Suite (Mids). Mids infers behavioral models of a component-based software system from the system's **execution traces** using a passive automata learning approach called Constructive Model Inference (CMI) [18]. The learned models can be compared and have their differences visualized on six levels of granularity, allowing a developer to explore the differences of two software versions at various abstraction levels [17]. Figure 1.2 shows an example of how Mids visualizes behavior models and their differences. The structural model comparison implementation of Mids, gLTSdiff, is open-source and will be available at `https://github.com/TNO/gLTSdiff`[2].

Mids can complement other quality assurance techniques such as software tests, benchmarks, or static analysis. Its only requirement is that the software system (or its runtime environment) can be modified to output a trace of its actions and communication. This can be a simple text-based trace as shown in Figure 1.3. The trace-based approach has several advantages: It is **implementation independent**, meaning the analysed code does not have to be written in a particular programming language or use a particular framework. While Mids makes some assumptions about its input traces (cf. [18]), the applied methods are not limited to ASML software; In this work, we use Mids to

[2] At the time of writing, gLTSdiff is not yet public.

```
0.022017363 lib_url_c > Curl_resolver_cancel !m13889
0.022017364 lib_asyn_h > Curl_resolver_cancel ?m13889
0.022018415 lib_asyn_h > destroy_Async_data !m13890
0.022018416 lib_asyn_thread_c > destroy_Async_data ?m13890
0.022019477 lib_asyn_thread_c < destroy_Async_data !r13891
0.022019478 lib_asyn_h < destroy_Async_data ?r13891
0.022020539 lib_asyn_h < Curl_resolver_cancel !r13892
```

Figure 1.3: Excerpt of a simple text-based execution trace. The data fields are, from left to right: Timestamp, component name, event type (“>” for Entry, “<” for Exit), function name, message identifier (“?” for received message, “!” for sent message).

analyse traces of the open-source, non-component-based software cURL[3], from which the trace in Figure 1.3 originates. Using Mids does not require additional **engineering effort** and it enables observing the system’s behavior in **real-world scenarios**.

1.2 Finding Timing Behavior Changes

Changes in the timing of a controllers’ actions and slow-downs caused by software bugs can have high impact on the throughput of a machine. Due to a component-based system’s complexity, such changes can propagate through multiple components, often taking adverse effects not at the location where they originated, but in an entirely unrelated location. Even just detecting that a software change has caused a change in timing is difficult and may only happen later in the development cycle. When a regression is detected, reproducing them in an isolated environment and finding the root cause can often be exceedingly difficult. Engineers told us of one case where a single character code change, which had caused a 200 ms delay, took the team two days to find.

While so far the focus for Mids has been on the *functional* requirements of a component-based software system, *non-functional* aspects such as timing have received less attention. The purpose of this work is to explore how to include timing information available in execution traces in the inferred system behavior models and in the model comparison, to help engineers find **significant changes in timing behavior**, and identify and prevent **performance regressions**. Ideally, points of interest in the time data should be automatically identified and presented to reduce the amount of data an engineer has to look at manually, similar to what is already done for functional changes (cf. Figure 1.2).

In this work, we cover several aspects of collecting and analysing timestamp data in the context of system behavior models, including:

- **Extracting timing data.** We must extract timing data from the obtained traces and add them to the system behavior models in a suitable format.
- **Normalization.** The collected raw timestamps require normalization in order to prepare for the detection of significant timing behavior changes and visualization. We must handle cases where functional differences between software versions cause timing data to be incompatible.

[3] https://github.com/curl/curl, Accessed 2022-12-11.

- **Detection.** Automated means of detecting significant timing behavior changes can assist engineers by narrowing the scope of the analysis. Implementing this presents similar challenges to the analysis of classical program benchmarks as performed by benchmarking toolkits such as BenchmarkDotNet[4] or Criterion[5]. However, other challenges addressed here are specific to our approach; In particular, we must be able to handle data with low sample sizes because traces produced by real workloads often do not contain sufficient data for every part of the system.

- **Visualization and User Interface Design.** While automated tools are helpful, nothing can beat assessment by a human expert. This is especially true for our case, as the definition of "significant" timing differences is fluid and can differ between software systems, execution environments and engineers. We should present intuitive visualizations of the timing data to aid the engineers achieve their tasks quicker.

We create a prototype extension to Mids, Mids+Time, and evaluate our approach on small hand-written examples, on real-world traces with artificially injected faults, and on real-world traces with behavior changes observed in practice at ASML. In a proof-of-concept, we show that Mids+Time is in principle also applicable to non-component-based software, and can even detect timing changes which are not originally caused by the software, but affect the software's execution. We find that our tool enables easy and intuitive inspection for many timing-related behavior changes and receive positive feedback from engineers at ASML, who would like to use the approach in future software analyses.

1.3 Overview

The remainder of this work is structured as follows: First, we cover the necessary theoretical background in Chapter 2. In Chapter 3, we describe our approach and implementation and discuss design trade-offs. In Chapter 4, we evaluate our approach both on artificially modified traces as well as on real-word traces. Chapter 5 presents related work and compares prior approaches to ours. Finally, in Chapter 6, we conclude our work and outline opportunities for future research.

[4] https://benchmarkdotnet.org/index.html, Accessed 2022-11-22.

[5] http://www.serpentine.com/criterion/, Accessed 2022-11-22.

2 Background

This chapter covers the theoretical background of our approach. We define Finite State Automata, which serve as basis for the system models. We then briefly introduce Constructive Model Inference and other theoretical concepts behind Mids.

2.1 Finite State Automata

Automata (or *state machines*) are a versatile abstraction for modeling software behavior. We define a deterministic finite automaton (DFA) as a tuple

$$\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$$

where Q is a finite set of states, Σ is a finite set of symbols called the “alphabet”, $\delta : Q \times \Sigma \rightharpoonup Q$ is a partial transition function, $q_0 \in Q$ is the initial state and $F \subseteq Q$ is a set of accepting states [18, 35].

An automaton $\mathcal{A}$ accepts a language of words $L(\mathcal{A}) \subseteq \Sigma^*$, a set of finite sequences of symbols such that

$$w = \sigma_0\sigma_1...\sigma_n \in L(\mathcal{A}) \iff \delta\left(\ ...\ \delta\left(\delta\left(q_0, \sigma_0\right), \sigma_1\right) ...\ \sigma_n\right) \in F,$$

i.e., a word is accepted by the automaton iff, starting in the initial state q_0, a sequence of transitions exists such that after having consumed the full word, an accepting state is reached [18, 35].

We use automata to model the observable behavior of component-based software systems. We define that the language of a model automaton should approximate the set of logs that can be produced by a system.

2.2 Constructive Model Inference

The task of reconstructing automata from a sample of output words has been extensively studied, and a variety of different approaches exist [5, 19, 30, 32, 41]. They can be roughly divided into two categories: Active learning approaches, which learn by repeatedly making membership queries to the system under learning (SUL), and passive learning approaches, which infer automata from a set of positive examples along with a set of negative examples or refinement heuristics [5, 18, 41].

In practice, the active learning approach is difficult to implement, since the live system must be modified to answer membership queries, and suffers from scalability issues [18, 41]. The results from passive learning approaches on the other hand may not generalize well and may depend on the chosen heuristics [18].

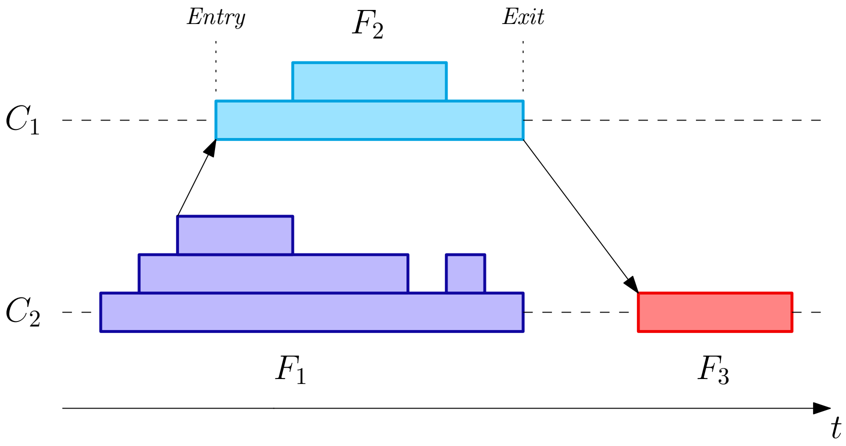


Figure 2.1: Example Timed Message Sequence Chart (TMSC) with two swimlanes. Boxes with the same color belong to the execution of one function ($F_1, ..., F_3$). Arrows represent messages between components (C_1, C_2). Stacked boxes indicate nested function calls. Every call starts with an *Entry* event and ends with an *Exit* event.

For the large-scale industrial setting at ASML, the Constructive Model Inference (CMI) approach has been devised; a passive learning method which makes extensive use of domain knowledge, in particular about the used Component-based Software Architecture, in order to construct models which align well with the mental model of the software authors [18]. In the following, we will first state some definitions and assumptions, and then briefly describe this method. More details can be found in the work of Hooimeijer et al. [18].

Service Fragments. A component-based software system consists of multiple components C_i, each of which provides a set of internal or external services. A service may in turn be split into multiple parts we call *service fragments* F_j. If, for example, a service defines a callback that is invoked after some asynchronous action finishes, then the calling function and the callback belong to the same service, but we view them as distinct service fragments. Each service fragment can be seen as a procedure which processes some internal or external message. It is assumed that components only execute one service fragment at a time and that service fragments can not be preempted [18].

System Execution Trace. It is further assumed that the provided CMI input, a system execution trace (or “log”), contains timed events, where each event corresponds to a message between service fragments. Events must come in pairs of an *Entry* event and an *Exit* event which must be properly nested. Note that the stated constraints are compatible with Timed Message Sequence Charts (TMSC) as introduced by Jonk et al. [22] and that CMI as well as the approach presented in this work use TMSCs as an input.

Figure 2.1 shows a TMSC visualization of a possible system execution trace. The x-axis represents time, the y-axis is divided into swimlanes corresponding to different components and events. Boxes fill the time between Entry and Exit events. The arrows between boxes represent dependencies, which are not explicitly contained in the trace, but are meant to visualize the communication between components.

Model Inference. Figure 2.2 shows an overview of the CMI process. An obtained system

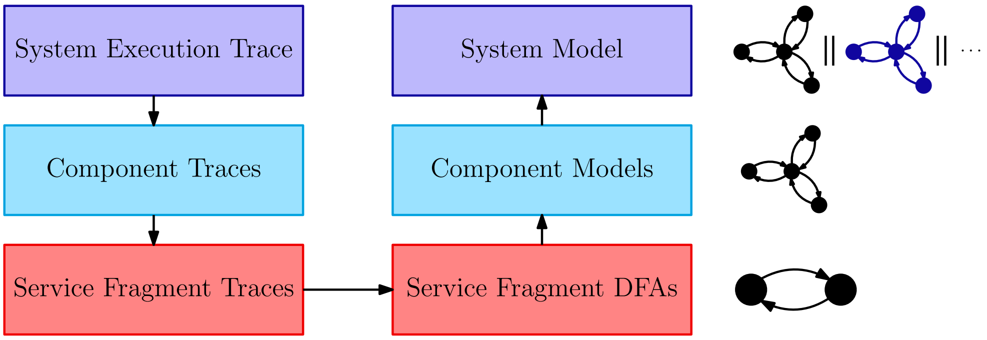


Figure 2.2: Constructive Model Inference process flow chart, based on [18].

execution trace is first split into individual traces for each component. These are then further split into service fragment traces. From the traces corresponding to one service fragment, a *Prefix Tree Automaton* is built. A Prefix Tree Automaton (PTA) is a tree-structured DFA that has a unique accepting state for each word in its language. This PTA is minimized and looped, meaning the terminal accepting states are merged with the initial state into one initial accepting state. The looping step implements the assumption that a fragment must always run to completion and is then ready to be executed again at a later time.

With the last step, we have obtained one DFA per service fragment, which can be composed into one DFA per component, and then one DFA for the entire system, in the subsequent steps.

First, the service fragment DFAs of one component are composed into one component behavior automaton. This is achieved by merging all initial states of the service fragment DFAs into one. Intuitively, this results in a "flower-shaped" component automaton where the central initial state is the idle state of the component (cf. Figure 2.2). Whenever a message is received, it is handled by one of the service fragment DFAs, transitioning back into the idle state on completion. This modeling choice allows for a component to execute service fragments in arbitrary order and any number of times, making for a sound approximation of the actual systems' behavior. It does, however, allow for execution sequences that could not occur in the real system. For example, a service fragment handling the result of a request can never execute before the service fragment making that request in the first place. (See Section 2.4 for a description of such communication patterns within ASML software.) For this reason, the component automata can optionally be composed with a behavior automaton to incorporate more domain knowledge, to impose restrictions on the order in which service fragments can be called. This would correspond to multiple distinct idle states with separate "flowers".

The component automata are composed into one system model by using asynchronous composition, which utilizes finite FIFO buffer state machines to enable asynchronous communication between components.

The methodology described in the remaining parts of this work is based on, and extends, the models produced by CMI. The manner in which the automata are constructed imposes a number of restrictions on the structure of the resulting models. For example, each transition resembling an Entry event will have a matching Exit event transition, and Entry and Exit event transitions will always be properly nested; Automata will always have at least one initial transition and, as a consequence of the well-nestedness, at least a matching final transition.

2.3 State Machine Comparison

Having obtained automata for traces of different software versions, methods for finding and highlighting differences in the automata are needed. We first describe a method by Walkinshaw and Bogdanov [40] used to compute *difference automata* which highlight the structural differences between two automata. This method is used in one of the six levels of Hendriks et al.'s six-level model comparison approach which we detail thereafter [17].

2.3.1 LTSDiff Algorithm for Comparing Automata

Walkinshaw and Bogdanov present a method of structurally comparing two automata or any other labelled transition system (LTS) [40]. Given two automata, the *LTSDiff* algorithm first computes a similarity score for every pair of states, based on their incident transitions' symbols and (recursively) on the neighboring states' similarity. Next, most similar states are identified as so-called *landmarks*. Starting from these landmarks, the surrounding states are explored, step-by-step accepting the most similar states into a set of matched states.

The remaining unmatched states and transitions are the structural differences between the two automata. By combining the automata and coloring "added" states and transitions green, and "removed" states and transitions red, a *difference automaton* can be constructed. Figure 1.2 shows a difference automaton as rendered in Mids. By presenting structural changes in-line, they can be intuitively reasoned about.

Mids uses a generalized and improved implementation of LTSDiff called *gLTSdiff*, which will be available at `https://github.com/TNO/gLTSdiff`.[1].

2.3.2 Six-Level Comparison Method

Difference automata highlight structural differences at the lowest level of the learned models, down to individual transitions. When analysing changes between software versions, more high-level views can be better for first gaining an overview of the system and localizing points of interest.

Mids implements the six-level approach outlined by Hendriks et al. in [17]. Given multiple *model sets* containing a number of *entity models* (in our case: service fragment DFAs), different levels of details at which the models can be compared are proposed [17]. In order of the highest level to the most fine-grained one, these are:

Level 1: **Model Set Variants**
Shows whether model sets accept the same language, i.e. have identical behavior.

Level 2: **Model Set Variant Relations**
Shows language-inclusion relationships between model sets.

[1] At the time of writing, gLTSdiff is not yet public.

Level 3: **Model Set Variant Differences**
Shows the number of different entities for every pair of model sets.

Level 4: **Model Variants**
Shows whether entity models accept the same language.

Level 5: **Model Variant Relations**
Shows language-inclusion relationships between models.

Level 6: **Model Variant Differences**
Shows structural difference between entity models as difference automata, as previously described.

For our work, levels 1-3 are less relevant, because we limit ourselves to **two** model sets. This is done to reduce the complexity of our proof-of-concept implementation, and because, when dealing with timing data, changes in timing between two software versions can be intuitively described as being "faster" or "slower". Levels 1-3 add not much value when comparing just two model sets.

2.4 Communication Patterns in ASML Software

The controller software of ASML machines is built upon a middleware software that enables different kinds of communication patterns [16]. These are typical producer/consumer interactions that can be found in other component-based systems as well. They allow a client to make different types of remote procedure calls, including:

- **Blocking Call.** The client makes a request to the server and is blocked until it resolves.
- **Request/Wait Call.** The client makes a non-blocking request and continues with different work. At some later point, it sends a blocking `wait` call to the middleware. As soon as the previously requested task finishes, the middleware returns the result. If the requested task has already finished before the `wait` call, it returns immediately. This is similar to typical async/await patterns in programming languages like JavaScript or Rust.
- **Function Completion Notification (FCN).** The client makes a request, indicating a procedure that is to be called by the middleware once the result is available. This is similar to passing a callback function in languages like JavaScript.
- **Trigger.** The client sends a message to the server without expecting any reply. The middleware can also be configured with custom triggers, for example to periodically send a trigger message to a specific server.

On the server side, the middleware enables two ways of handling requests:

- **Synchronous Handling.** The middleware calls a service on the service which immediately handles the request and returns the result.

- **Asynchronous Handling.** The middleware notifies the server of the request. At a later point, the server may answer the request by calling an appropriate function of the middleware.

The middleware also allows for raising and subscribing to *events*. Raising an event invokes a callback on every subscriber.

Through the middleware, client and server are completely decoupled and opaque. The client does not need to know how the server handles its request, and the server does not need to know how the request was made or how the result is received.

3 Methodology

In this chapter, we describe our approach to analysing timing data, and its implementation in MIDS+TIME, our prototype extension to MIDS. As stated in the introduction, the goal of this work is to extract timing data from the system traces (Section 3.1), add it to the system models (Section 3.2), and help engineers find changes in timing behavior and performance regressions by means of both automatic analysis (Section 3.5) and visualizations (Section 3.6). We go in order from the most low-level representation, the traces, to the most high-level ones, the visualizations.

We assume that we are always given two traces as an input, a baseline *reference* trace from a *reference* system and a *changed* trace from a (possibly) *changed* system. The method works best when these are reasonably similar in structure, with only a limited number of functional changes, as we want to focus on timing changes. It is not a goal to investigate how functional changes and timing changes relate, though this could possibly be investigated in future work.

Having added timing data to the system models, there are countless imaginable ways of analysing and comparing the available data of two traces based on more or less domain knowledge: Analysing service fragment call timings, communication patterns, component up- and idle times, and many more. Based on feedback from engineers, in this section, we focus on raw service fragment execution time and on the timing of individual function calls within service fragments. We seek to extend Levels 4 and 6 of MIDS' six-level approach (cf. Section 2.3.2) with this data, to visualize timing differences in a way that is intuitive, understandable and explorable.

3.1 Extracting Timing Data from Execution Traces

In the previous chapter, we have described how we learn DFAs from the output trace of a component-based software system. The transition symbols in these automata correspond to messages exchanged between service fragments. The language which is accepted by the learned automaton is an approximation of the set of output traces which could be generated by the system.

In order to analyse the timing behavior of the system, we associate a sequence of *absolute timestamps* with each transition of the automaton, corresponding to the times at which the event ocurred.

We define a Time-Annotated Deterministic Finite Automaton (TDFA[1]) as follows:

$$\mathcal{A} = (Q, \Sigma, \delta, q_0, F, T)$$

[1] Note that Hooimeijer et al. [18] also define a "TDFA", which is not to be confused with our definition.

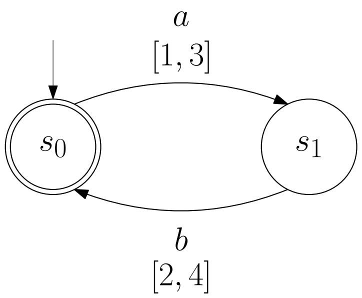


Figure 3.1: Example TDFA.

where $Q, \Sigma, \delta, q_0, F$ are defined as in Section 2.1 and

$$T : Q \times \Sigma \rightharpoonup \mathbb{R}_+^+$$

maps transitions to finite sequence of non-negative timestamps in chronological (i.e., ascending) order. We also call these sequences *time samples*. Because of the way the automata are constructed from the input trace, the event represented by a transition has ocurred at least once, which is reflected in the fact that the time samples must be non-empty.

Example. Figure 3.1 shows an example TDFA with $Q = \{s_0, s_1\}$, $\Sigma = \{a, b\}$, $\delta(s_0, a) = s_1, \delta(s_1, b) = s_2$, $q_0 = s_0$, $F = \{s_0\}$, and

$$T(s_0, a) = [1, 3]$$
$$T(s_1, b) = [2, 4].$$

Event a occured at time 1 and 3, event b at 2 and 4.

The semantics and accepted language of a TDFA $(Q, \Sigma, \delta, q_0, F, T)$ are identical to a DFA $(Q, \Sigma, \delta, q_0, F)$.

3.1.1 Learning a TDFA from Trace and DFA

Given a system execution trace w with timestamps t and a DFA learned from that trace as described in Section 2.2, we can add time annotations to the DFA to yield a TDFA by simply simulating the transitions of the automaton one-by-one and adding the event timestamps as the transitions are taken (see Algorithm 1).

Algorithm 1 Adding time annotations to a DFA

```
Input: DFA $\mathcal{A} = (Q, \Sigma, \delta, s_0, F)$, event trace $w \in \Sigma^*$, event timings $t \in \mathbb{R}_+^+$, $|w| = |t|$.
  $T \leftarrow \text{dom}(\delta) \times \{\,[\,]\,\}$                ▷ Initialize all transitions with empty time sequences.
  $s \leftarrow s_0$
  for $i \in \{0, ..., |w| - 1\}$ do
      $T(s, w_i) \leftarrow T(s, w_i) + [t_i]$
      $s \leftarrow \delta(s, w_i)$
  end for
Output: TDFA $\mathcal{A}' = (Q, \Sigma, \delta, s_0, F, T)$.
```

Trace 1

```
11 component > f
12 component > g
13 component < g
14 component < f
```

Trace 2

```
81 component > f

83 component > g
84 component < g
85 component < f
```

Trace 3

```
81 component > f
82 component > h

84 component < h
85 component < f
```

Figure 3.2: Three minimal example traces for Figure 3.3 and Figure 3.4. The data fields in each line are, from left to right: Timestamp, component name, event type (“>” for Entry, “<” for Exit), function name. For readability, we do not use full Unix timestamps. Trace 2 is functionally equivalent to Trace 1, but the Entry event of the call to g has been delayed by one unit. In Trace 3, the calls to g have been replaced by calls to h, and the Exit event of h is delayed by one unit compared to the Exit event of g in Trace 1.

Conversely, we can recover the original trace from a TDFA, if the event timings are strictly monotonic, by always taking the transition with the minimal next higher timestamp.

3.2 Normalization

With the previous step, we have obtained a system model consisting of one TDFA for each service fragment. The traces we process contain absolute time values in nanoseconds since the Unix epoch (00:00:00 UTC on January, 1st 1970). These carry over to the models, so each transition is annotated with a sequence of nanosecond-precision Unix timestamps.

This representation is useful for some analyses, for example to compute durations between two arbitrary events, such as start and end of a service fragment. However, for other tasks, in particular on service-fragment level, this representation is not useful or disadvantageous. We want to enable the engineer to assess the performance of the system, and for that, *durations* are much more useful than discrete points in time. Strictly speaking, the Unix timestamps represent durations, but durations with more recent starting points are much easier to interpret and compare: For example, the elapsed time since the start of the system's execution, or the elapsed time since the very last function call. Smaller numbers are also more readable than full nanosecond-precision Unix timestamps (e.g. `845032054749200`).

Consider the artificial example traces shown in Figure 3.2. Figure 3.3 contains four examples of different normalization strategies (Absolute, Relative, Stack-based, Service-Fragment Absolute) visualized on example automata corresponding to traces 1 and 2. The traces differ in that **a one unit delay** has been added to the second event in Trace 2. Figure 3.4 contains the same normalization strategies, this time applied to traces 1 and 3, where trace 3 contains a structural change (calls to g have been replaced by calls to h) and additionally a one unit delay in the third event. The annotated red numbers on the transitions correspond to T_L, the timestamps from Trace 1. The green numbers correspond to T_R, the timestamps from the changed trace. The functional difference in Figure 3.3 is represented by green and red transitions in the usual difference automata format. For brevity, we write an up-arrow ↑ for Entry events and a down-arrow ↓ for Exit events, so the first transition in the left automaton of Figure 3.3 corresponds to the log line “`11 component`

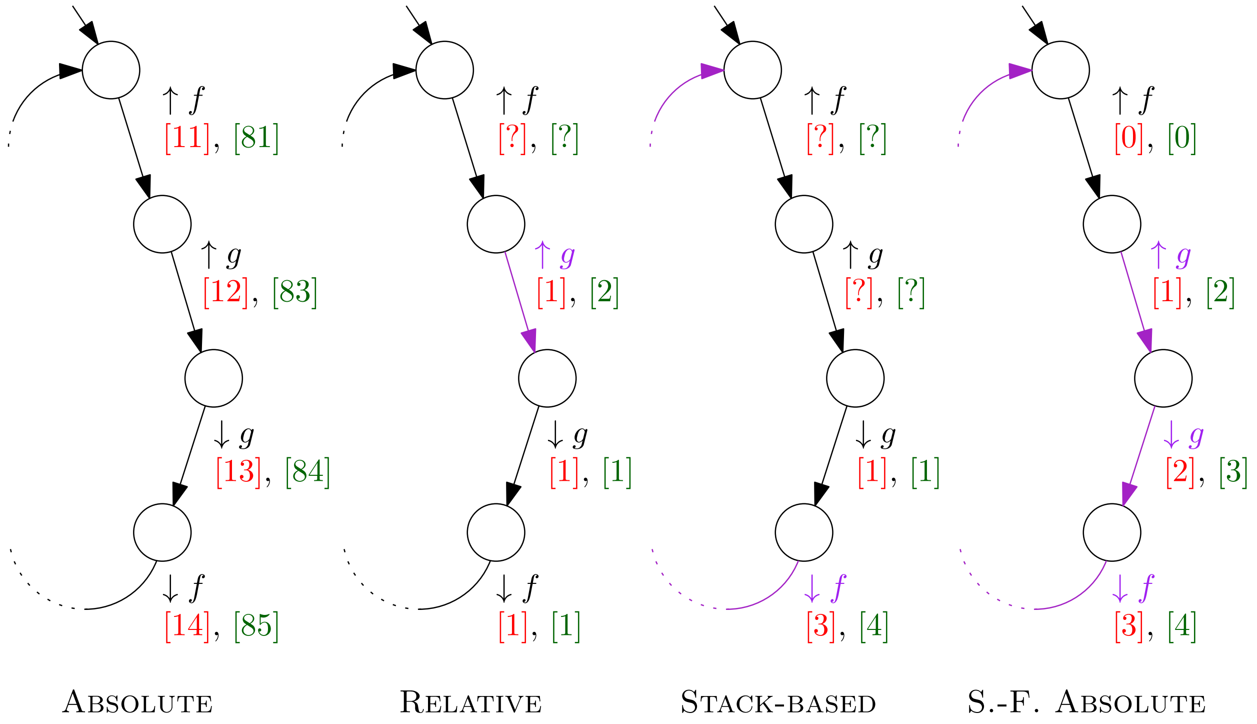


Figure 3.3: Example difference automata based on Figure 3.2's trace 1 and 2. The red and green numbers represent T_L and T_R, respectively. Detectable timing differences between T_L and T_R are highlighted in purple.

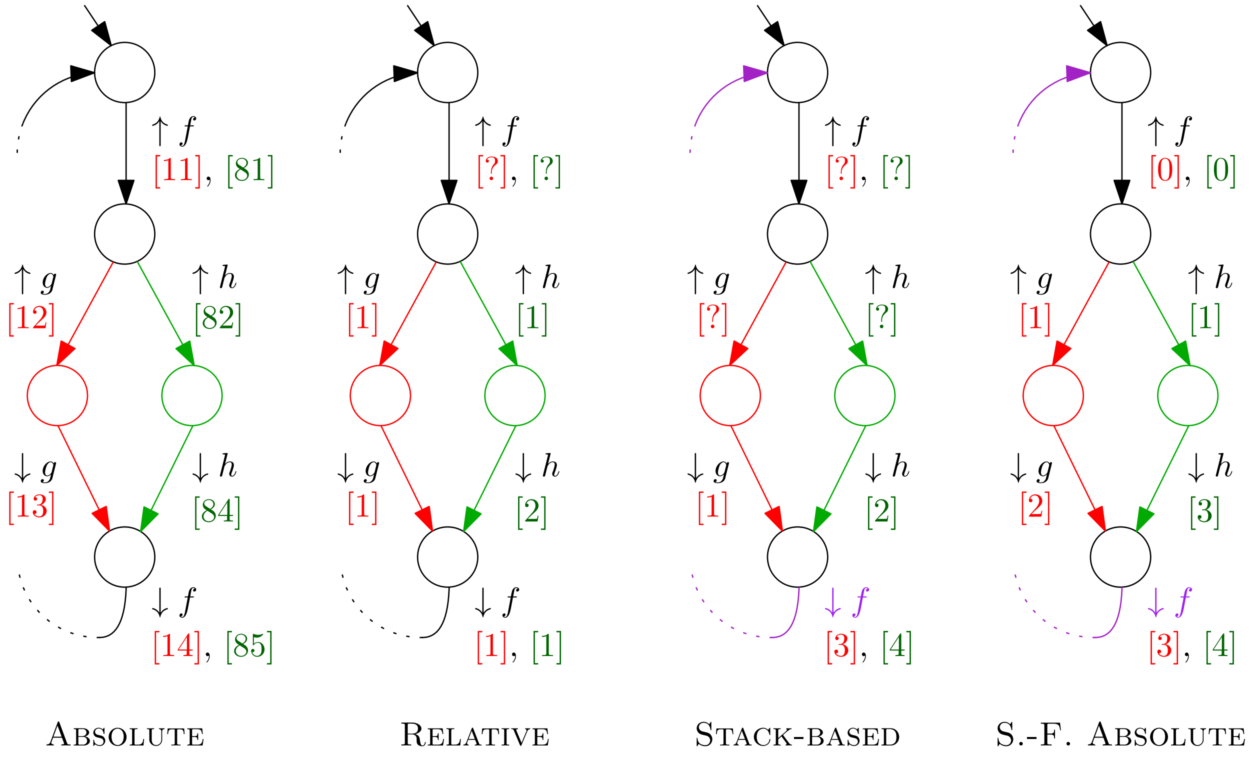


Figure 3.4: Example difference automata based on Figure 3.2's trace 1 and 3. In this example, a functional difference is present, which is highlighted by the red and green transitions. These transitions have only timing values for either T_L or T_R, because their events are only present in one of the two traces.

Property	Absolute	Relative	Stack-Based	S.-F. Absolute
Easy-to-understand transition labels?		✓	✓	
Timings intuitively comparable?		✓	✓	✓
Comparatively noise robust?			✓	✓
Intuitive fault locations?		✓		✓

Table 3.1: Overview of advantages and disadvantages of the presented normalization strategies.

`> f`”. Timing differences between T_L and T_R are highlighted with purple transitions and transition labels, which is how we highlight timing difference on difference automata in Mids+Time.

Absolute. The left-most automata in Figures 3.3 and 3.4 contain no time value normalization, the annotated timestamps are the raw, absolute timestamps from the traces. As previously mentioned, these time values are difficult to compare, because their baseline is unknown. The unit delay in Figure 3.3 is not cleary apparent. In this simplified example with small absolute timestamps, it may be possible to guess the relationship between the timestamps in the first and the second trace and conclude that a delay occurred, but this is infeasible in practice. The situation in Figure 3.4 is similar.

Relative. The Unix epoch as a baseline is one extreme; the other extreme is shown in the second automata in Figures 3.3 and 3.4. Here, transitions are labelled with the time elapsed since the most recent event on the same service fragment. For example, the second transition in the automata of Figure 3.3 has the annotations $1 = 12 - 11$ and $2 = 83 - 81$. Note that the first transition of the service fragment is a special case, because there is no previous event. In Figure 3.3, the timing difference is very clearly apparent at the exact location where it occured ($\uparrow g$, $T_L = [1]$ versus $T_R = [2]$), and relatively large (the time value is larger by a factor of 2). The annotated values of this normalization strategy are also mostly intuitive – they are simply the time “since the last transition”. It may not be quite as intuitive in all situations though, because two subsequent events may be unrelated from an engineer’s perspective. For example, the time values on the third transition from the top in Figure 3.3 give the time between Entry and Exit of the call to g ($\uparrow g$, $\downarrow g$), i.e., the runtime of function g. However, the time values on the fourth transition represent the time elapsed between the two exit events ($\downarrow g$ and $\downarrow f$), which has no intuitive interpretation. Another problem is that the method performs poorly on traces with functional changes, as illustrated in Figure 3.4: The timing difference is only present on the second green edge, where T_L is empty and cannot be compared against. While in this small example, we can easily see which green and red transitions must match and conclude that a timing difference is present between $\downarrow g$ and $\downarrow h$, this kind of matching of transitions is infeasible in larger systems with many functional differences. We will see how other strategies solve this issue next.

Stack-based. In the “Stack-based” approach, every Exit transition is labelled with the time since its corresponding Entry event. This results in intuitive annotations, which allow easy reading of the functions' runtimes. In Figure 3.4, we can see that the Stack-based approach can also handle functional changes when they affect the runtime of the enclosing function call: The unit delay is visible of the Exit edge of f, because f's runtime has increased. The relative size of the delay is smaller than with the Relative normalization strategy (4/3× in both Figures 3.3 and 3.4, because the runtime of g was increased by 1/3). This generally holds true, making this strategy more robust against small variations compared to the Relative approach. While the Stack-based strategy is practical and intuitive for the task of inspecting service fragment runtimes, it is limited to that, and has several drawbacks. One is that it is not clear how the Entry events ($\uparrow f$, $\uparrow g$, $\uparrow h$) should be labelled. Another is the fact that the unit delay in Figure 3.3 shows up in an unexpected location: At the Exit event of f. While this is factually correct, since the delay only affected the outer service fragment f's runtime, it is unintuitive, because it did affect the execution timing of g in that its start was delayed. This information is lost with this normalization strategy. The visualization also makes it appear that the delay occurs later in the execution than it actually does.

Service-Fragment Absolute. The fourth strategy, which we call Service-Fragment Absolute, strikes a balance between intuitive annotations and robustness. Here, the time values represent the time elapsed since the start of the service fragment, in this case the Entry event of f. While this may not be immediately intuitive, the semantics can be understood quickly, and it comes with a number of advantages: It is more robust against noise, because small changes impact the total service fragment runtime less than the timing of a single event. Delays are localized close to their cause; In Figure 3.3, the delay can be traced to $\uparrow g$, which is the first transition colored purple and represents the first delayed event in the trace. The propagation of a change to successive transitions, which seems to be a disadvantage of this strategy at first, can actully be an advantage. It helps to find small but repeating delays, which accumulate across the runtime of a service fragment. It also helps detect cases where the timings between versions are briefly different, but later align again. This would result in parts of the automata being highlighted purple, becoming black again at later transitions. The propagation also helps in handling functional differences. This can be seen in Figure 3.4, where the unit delay propagates from the section with functional changes to the black $\downarrow f$ transition where both traces match again, and where the timings can be compared.

The advantages and disadvantages of the presented normalization strategies are summarized in Table 3.1. For Mids+Time, we implement all four strategies, but due to the stated advantages, we decide to mainly use the Service-Fragment Absolute approach in practice. It is used for all of our evaluation (Chapter 4). In principle, it is possible to define arbitrary normalization strategies, suitable to the task at hand. All following methods in this chapter are in general independent of the used normalization strategy.

3.3 Computing Difference Automata from TDFA Pairs

After the previous processing steps, we have two system models consisting of a set of TDFAs. Most service fragments should have corresponding TDFAs in each of the two system models. If a service fragment was removed or added in a software change, it may only have a corresponding service fragment in one of the sets. We do not consider these cases in the following, as there is no timing data to directly compare. For all other service fragments, there is a **pair** of two TDFAs with different timing annotations, and, possibly, functional differences.

At this stage, Mids applies the gLTSdiff implementation of Walkinshaw and Bogdanov's algorithm to combine two automata into one difference automaton (see Section 2.3.1). This process necessitates matching corresponding states and transitions of the two automata and then merging the matched states and transitions into one. We must modify the algorithm in order to correctly handle the combination of transitions, specifically our time annotation function T. We define that the transitions of a difference automaton of two TDFAs should be annotated with an ordered pair of possibly empty time samples, where the first entry corresponds to time values of the reference trace and the second entry to time values of the changed trace:

$$T_{\text{diff}} : Q \times \Sigma \rightharpoonup \mathbb{R}_+^* \times \mathbb{R}_+^*$$

This results in a difference automaton where all matched "black" transitions have a pair of time samples

$$(T_L, T_R) = ([t_{L,1}, ..., t_{L,n_L}], [t_{R,1}, ..., t_{R,n_R}]) \in \mathbb{R}_+^* \times \mathbb{R}_+^*, \; n_L, n_R \in \mathbb{N}.$$

(Note that $_L$ and $_R$ stand for "left" and "right". T_R does *not* contain the timings of the reference trace.) Unmatched "red" and "green" transitions will have pairs where one entry is empty, i.e., $(T_L, [\,])$ or $([\,], T_R)$. This principle was already implicitely shown in the difference automata of Figure 3.4.

3.4 Derived Timing Statistics and Metrics

At this stage in the analysis process, we have obtained difference automata which model the differences and commonalities between the reference system and the changed system. The commonalities are represented by matched states and transitions. The matched transitions are labelled with pairs of time samples (T_L, T_R) that lend themselves to comparison and can answer questions like

- "Was this function call reached *earlier* in the new software version compared to the reference version?"
  For example: $T_L = [2, 3, 2], T_R = [1, 2, 1]$: All time values in the changed trace are smaller, indicating a possible speed-up. The interpretation of this speed-up depends on the chosen normalization strategy.

- "Has the timing pattern of this function call changed?" (It is called twice as often, in shorter intervals, or similar.)
  For example: $T_L = [1, 5, 1, 5, 1, 5], T_R = [3, 3, 3, 3, 3, 3]$: In the reference trace, the time value distribution has two modes, one and five, while in the changed trace, they are all three. This may indicate a timing behavior change.

- "Is this function call slower to return, indicating a performance regression inside the called function?"

For example: $T_L = [2, 3, 2], T_R = [1, 2, 1]$ on the Exit event of a function with the Stack-based normalization strategy.

The transitions of difference automata are however not the only place where we encounter pairs of timestamp samples. We can derive additional timing metrics by computing the durations between specific events or event types. One simple example is the runtime of single service fragments. This can be computed by calculating the elapsed time between the start and the end events of a service fragment. Because events may occur multiple times in one execution (a service fragment may be called multiple times), we have to always compute pair-wise differences between a start event and the corresponding *next*, i.e., smallest larger, end event. Consider the following Absolute time annotations for events of a service fragment f: $T(s, \uparrow f) = [10, 20, 30, 40]$ and $T(s', \downarrow f) = [15, 26, 36, 45]$. The resulting execution times are

$$T_{\text{exec}} = [15 - 10,\ 26 - 20,\ 36 - 30,\ 45 - 40] = [5, 6, 6, 5].$$

The result is again a sequence of time values, to which the same detection heuristics and visualizations can be applied as for the TDFA's timing annotations. Other simple-to-compute metrics beyond service fragment execution times can be: Service fragment downtime, i.e., time spans where the service fragment was inactive; component downtime, i.e., downtime between service fragments on the same component, and the inverse, component uptime. Theoretically, we could even allow engineers to define custom metrics by letting them choose arbitrary pairs of transitions. This has however not been implemented in our Mids+Time prototype.

Further application-specific metrics can be derived from the communication patterns which we have introduced in Section 2.4. Based on communication metadata that is contained in some of the traces supplied to us, we can compute:

- **Blocked Time.** Time spent waiting on a blocking call.
- **Waited Time.** Time spent waiting after a `wait` was invoked on a Request/Wait call.
- **Time waited for FCN callback.** Time between issuing an FCN call and the result callback being invoked by the middleware.
- **Synchronous Handling Time.** Time spent synchronously handling a request.
- **Trigger Handling Time.** Time spent synchronously handling a trigger message. This is measured as the time between receiving the trigger message from the middleware and the handling function returning.
- **Event Handling Time.** Time spent synchronously handling an event (This is not to be confused with a trace event). Measured in the same way as Trigger Handling Time.
- **Asynchronous Handling Time.** Time between an asynchronous call was received and being replied to. Because asynchronous handlers do not need to handle the request immediately, this may include time spent waiting on other services or time that was spent doing unrelated work.

These statistics can be useful to analyse the communication behaviour and the timing of the communcations, to enable an engineer to answer questions like: “Does this service fragment spend the same amount of time waiting for a reply from another component as before the software change?”, “Has the time spent handling triggers increased?”, and more.

The engineers we collaborated with for our evaluation were mainly interested in the raw runtime of service fragments, and less in the Communication Pattern analysis. For this reason, while we still include this metrics and visualize them in Mids+Time (see Section 3.6.4), we do not include them in the evaluation. Other groups may may still be interested in analysing the timings of component communications though.

3.5 Automatic Timing Change Detection

The goal of this section is to investigate how we can automatically detect timing changes between two time samples, independenly of how they were obtained, and how we implement these mechanisms in Mids+Time.

3.5.1 Challenges in Automatic Timing Change Detection

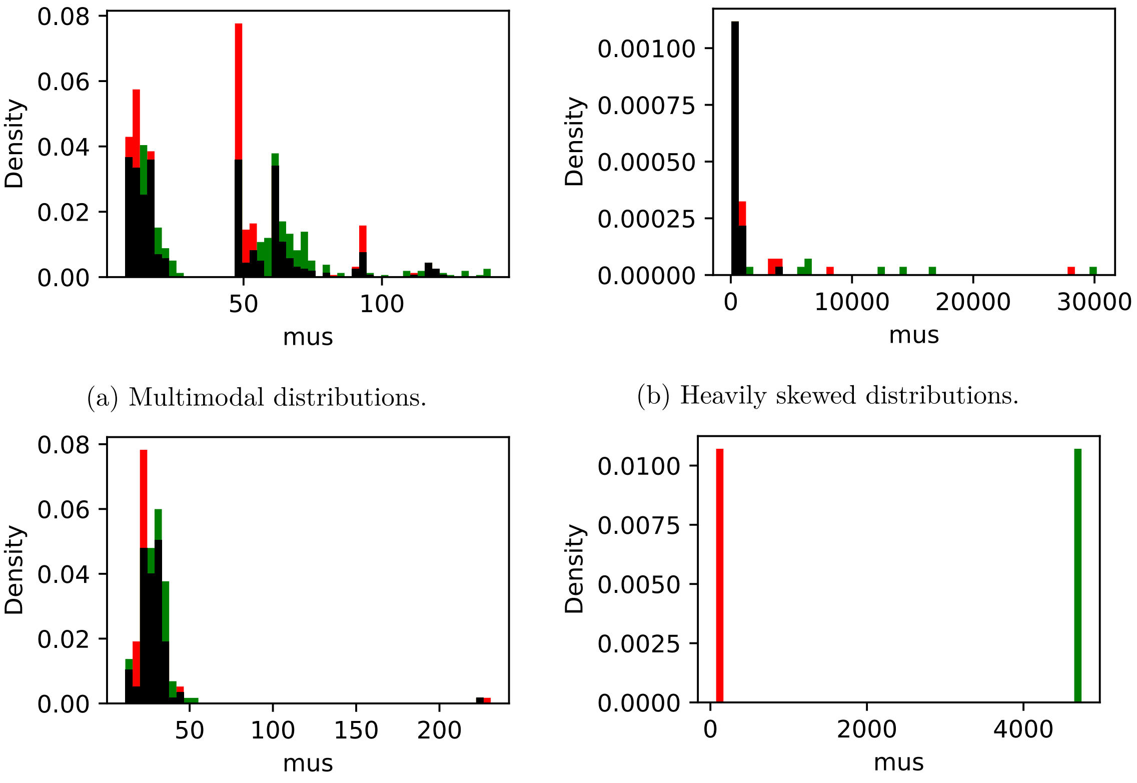
In the introduction, we motivate why finding timing behaviour changes and timing regressions is important, but we do not clearly define what a *significant* timing behavior change is. Doing so is difficult, because the point at which two sets of time samples should be considered the same or significantly different is both fluid and subjective. It changes from system to system, from engineer to engineer, and from use case to use case. What is a large delay in one component may be disregarded as noise in a different component. Automatic approaches can thus never fully replace a human’s assessment of the timing data. Coffin et al. write on this matter:

> “Exploratory data analysis is an invaluable tool [...]. Simple graphs and summary statistics, while not an end in themselves, often suggest which models, tests, and intervals will most clearly answer the experimenter’s questions.” — Coffin et al., [9].

For this reason, we attempt to both apply heuristics to find significant changes (described in this section), but also provide the engineers with helpful visualizations for manual inspection (described in Section 3.6). Our heuristics are parametrized, meaning they contain parameters such as fixed significance thresholds which can and should be adjusted by the user. For our evaluation, the parameters were tuned according to feedback by the company’s engineers, but they can be changed to fit other use cases without much additional effort.

Beyond the challenge of the sometimes fuzzy definition of significant timing behaviour changes, we need to deal with challenges which also arise in classical performance benchmarking and benchmark evaluation [1,3,9,14]:

- **Noise.** Noise is mostly unavoidable when measuring timing data. The detection methods should be robust in that small deviations in the data should ideally not change the heuristics’ predictions.



(a) Multimodal distributions.

(b) Heavily skewed distributions.

(c) Distribution with large outliers.

(d) Low sample size distribution ($n = 1$)

Figure 3.5: Different examples for non-normal service fragment execution time distributions. (Red: distribution of reference trace, green: distribution of changed trace, black: overlap.)

- **Non-Normality.** Many statistical methods, for example the standard two-sided *Student's t-test*, or similarly *Tukey's range test*, require the underlying distribution(s) to be normal [23, 38]. This assumption often does not hold for performance data [1, 3, 13, 14]. This is illustrated by the examples in Figure 3.5, which are all sourced from real-world data. We observe multiple peaks (*multi-modality*, 3.5a), skewed distributions (3.5b) and large outliers (3.5c) in our data.

- **Low Sample Sizes.** This problem is specific to our application and worsens the two previous problems. In systems where a software units' performance can be measured in isolation, as is done in benchmark frameworks such as BenchmarkDotNet[2] or Criterion[3], one can easily influence the number of measurements. Repeatability is in fact a key requirement for reliable performance measurements [1]. We, however, work with real-world traces capturing real-world work loads, and have no control over the sample sizes. As a consequence, we often have to deal with low sample sizes, such as shown in Figure 3.5d. Figure 3.6 shows a typical distribution of the number of times individual service fragments have been called over the course of one execution. As can be seen, it follows an exponential distribution, where many service fragments are called only a few times (1-100) and few service fragments are called

[2] `https://benchmarkdotnet.org/index.html`, Accessed 2022-11-22.
[3] `http://www.serpentine.com/criterion/`, Accessed 2022-11-22.

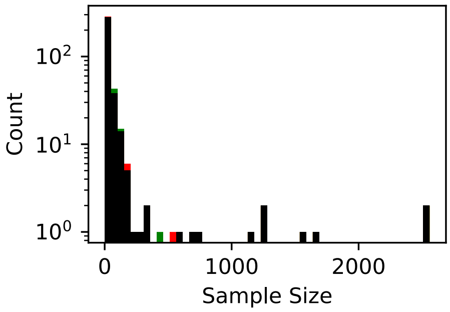


Figure 3.6: Service fragment call count distributions from a pair of real-world traces. (Red: distribution of reference trace, green: distribution of changed trace, black: overlap.)

many times (>1000). We observed similar call count distributions in non-industrial traces, for example on an experiment we conducted with the open-source software cURL[4]. This intuitively makes sense, as software often contains functions coordinating some workflow (like a "`main`"-function) which are only called once or few times, while other functions, like functions providing some logging functionality, are called often and from many places. Low service fragment call counts imply that the transitions inside these service fragments are also visited only a few times. Our detection heuristics must still be able to perform well in these cases.

Because no one detection heuristic solves all these challenges adequately, we decide to implement multiple, simple heuristics and combine them into one detection mechanism. In Chapter 4, we evaluate these detection methods both independently and in combination.

3.5.2 Threshold-Based Detection Methods

We find that for low sample size time samples, where statistical tests are not applicable (e.g., $n < 10$), simply comparing the sums of the time samples can be sufficient. Time samples are classified as significantly different when either their absolute difference exceeds a fixed threshold c

$$\left| \sum_i t_{L,i} - \sum_j t_{R,j} \right| > c \qquad (\textsc{Abs})$$

(in our work, $c = 6\,\text{ms}$ proved to be a good threshold), or the ratio between the two sums exceeds a fixed ratio r:

$$\max \left\{ \frac{\sum_i t_{L,i}}{\sum_j t_{R,j}}, \frac{\sum_j t_{R,j}}{\sum_i t_{L,i}} \right\} > r \qquad (\textsc{Rel})$$

If, for example, $r = 1.5$, time samples will be classified as significantly different if their sums differ by a factor of more than 1.5×.

[4] `https://curl.se/`, Accessed 2022-01-16.

3.5.3 Hypothesis Significance Test-Based Detection Methods

For samples with larger sample sizes, we can use hypothesis significance testing. As our definition of a significant timing change is not strict, a number of different null hypotheses appear as valid options to test against.

In their study about performance regressions, Chen at al. use *Student's t-test* to determine how likely two samples are drawn from distributions with the same mean [8]. However, the t-test requires normally distributed samples or a large sample size [23], which we can often not guarantee. Coffin et al. recommend using *Friedman's test* for ranking the performance of n different algorithms [9]. This is also used by FuzzBench[5], a service by Google for evaluating the performance of fuzzing software. Friedman's test is not useful to us, as we only compare $n = 2$ samples, but we consider the *Mann-Whitney U* (MWU) test, which is also a rank-based test, but for only *two* independent samples [24]. The null hypothesis of the MWU test is that no one underlying distribution is stochastically greater than the other. BenchmarkDotNet[6] also uses the Mann-Whitney U test for samples of size at least 3. Akinshin recommends either the MWU test or the non-parametric Two-Sample *Kolmogorov-Smirnov* (KS) test for evaluating performance statistics [3].The KS-test considers the shape of the empirical distributions in addition to their relative location: Let F_1, F_2 be the empirical cumulative distribution functions of the samples, calculated as

$$F_i(t) = \frac{1}{n_i} \sum_{j=1}^{n_i} \mathbf{1}\{t_{i,j} \leq t\}, \qquad i \in \{L, R\}$$

then the null hypothesis that the two samples are drawn from the same distribution is rejected if the maximum absolute difference between F_L and F_R exceeds a certain critical threshold [11]. The two-sample KS test effectively uses the greatest distance between the two empirical cumulative distribution functions as a test statistic; This statistic is both independent of the underlying distributions and robust for low sample sizes [7, 26].

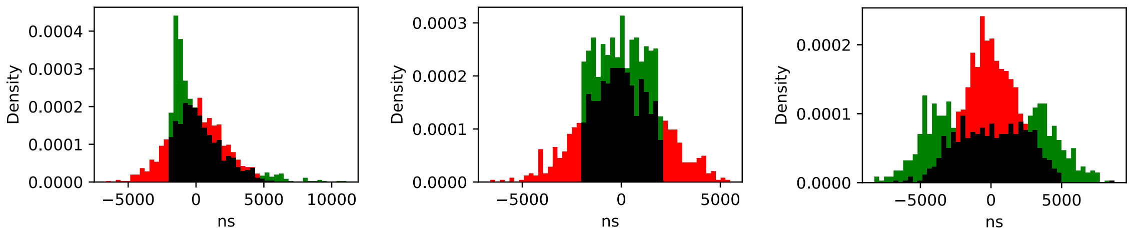
Figure 3.7 shows three examples where the green and red distributions are dissimilar, but share roughly the same mean, which results in the t-test not rejecting the null-hypothesis. Similarly, the MWU test does not reject its null-hypothesis because due to the symmetry, no one distribution is clearly stochastically greater or smaller than the other. With the KS test, the distributions are recognized as significantly different ($p < 0.05$). Conversely, we can also construct examples where visually different distributions are detected by the MWU test, but not by the KS test (see Figure 3.8).

We find both the MWU test and the KS test to perform well on real world data and decide to include both in the evaluation.

In practice, the null hypothesis is often rejected even if the absolute magnitude of the timing change is very small. To filter out false positives with negligible real-world impact, we combine the hypothesis tests with threshold-based detection methods (Section 3.5.2) in our tool. This is described in more detail in Section 3.5.5.

[5] https://github.com/google/fuzzbench, Accessed 2022-01-02.

[6] https://benchmarkdotnet.org/index.html, Accessed 2022-11-22.

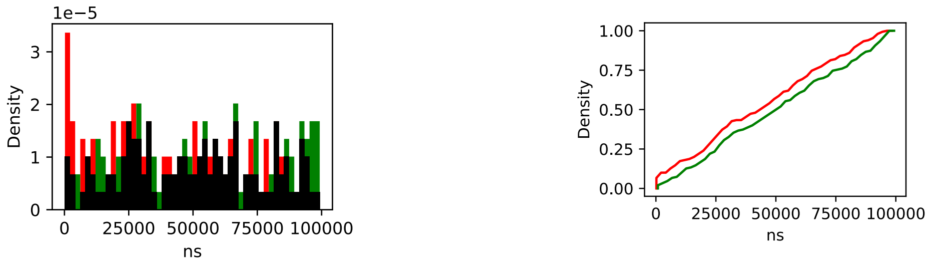


(a) $p_{\mathrm{KS}} \approx \mathbf{1.6 \cdot 10^{-14}}$, $p_{\mathrm{MWU}} \approx 0.33$ $p_{\mathrm{T}} \approx 0.67$

(b) $p_{\mathrm{KS}} \approx \mathbf{2.2 \cdot 10^{-12}}$, $p_{\mathrm{MWU}} \approx 0.63$ $p_{\mathrm{T}} \approx 0.98$

(c) $p_{\mathrm{KS}} \approx \mathbf{0}$, $p_{\mathrm{MWU}} \approx 0.89$, $p_{\mathrm{T}} \approx 0.63$

Figure 3.7: Sample distributions with significant differences are detected ($p < 0.05$) only by the K-S test, not by the t-test or the Mann-Whitney U test. In all cases, the means of the distributions are close, but the shapes of the distributions are dissimilar – which is not considered by the t-test. Due to the distributions symmetry, the null hypothesis of the Mann-Whitney U test is not rejected.



(a) Histogram.

(b) Empirical CDF of (a).

Figure 3.8: Example for (artificially produced) samples that are detected as significantly different ($p < 0.05$) by the Mann-Whitney U test, but not by the K-S test ($p \approx 0.63 > 0.05$). This is likely due to outlieres at the lower edge, which causes MWU to rank the red distribution lower, while the empirical cumulative distribution functions (CDFs) shown on the right are too close for the K-S test to consider the distributions difference. In practice, a human would probably not perceive these distributions as significantly different.

3.5.4 Quantile Shift-Based Detection Method

In addition to the low sample size threshold-based detection methods and the hypothesis-test-based methods for samples with higher samples sizes, we experiment with another detection method which works for low and high sample sizes. It is motivated by the goal to detect cases where the underlying distribution of T_R was clearly and unambiguously shifted in **one direction** (either positive or negative) compared to T_R. A small example for a positive shift would be: $T_L = [1, 5, 1, 5, 1, 5], T_R = [2, 6, 2, 6, 2, 6]$. Inspired by the KS test and the Shift Function visualization, which is presented later in Section 3.6.3, we want to check whether the quantiles of one sample are greater or equal (resp. less or equal) than their respective quantiles in the other sample:

$$\forall p \in \{0.1, 0.2, ..., 0.9\} : Q_{L,p} \geq Q_{R,p} \qquad \text{or} \qquad \forall p \in \{0.1, 0.2, ..., 0.9\} : Q_{L,p} \leq Q_{R,p}$$

Here, $Q_{L,p}$ denotes the Harrell-Davis quantile estimate for the p-quantile of T_L [15]. We use the Harrell-Davis quantile estimator, as opposed to just linearly interpolating adjacent sample elements as is done in some cases by other benchmarking frameworks (e.g. Criterion.rs[7]), because it provides quantile estimates with higher statistical efficiency [3, 15]. Note that looking only at the deciles ($p = 0.0, 0.1, 0.2, ...$) is an implementation choice. One could also choose to sample the quantiles at a finer resolution. We start from the 0.1-quantile and end at 0.9, dropping the extreme values of the samples, where the quantile estimation can become unstable [2].

If the left equation ($\geq$) holds, we conclude that the underlying distribution has shifted to the left, i.e., the timings got smaller; The opposite holds if the right equation ($\leq$) is fulfilled. For similar reasons as with the hypothesis-test-based methods, we again combine this Shift detection method with threshold-based methods in practice, to filter changes with small real-world impact.

3.5.5 Combined Detection Method

We combine the presented detection methods, which fulfill different purposes in solving the aforementioned challenges, into one detection heuristic. The details are visualized in Figure 3.9:

A pair of samples is classified as significantly different, if at least one of the following holds: (1.), the null hypothesis of a hypothesis test (either KS or MWU) is rejected with $p < 0.05$ and the absolute difference Abs is at least 0.1 ms or (2.), the Shift detection method concludes that one distribution is shifted and the absolute difference Abs is at least 0.1 ms or (3.), the absolute difference Abs is at least 6 ms. (1.) is intended as a noise-insensitive generally applicable detection method, (2.) is intended as a catch-all for samples that have been shifted in one direction, and (3.) is intended to detect huge deviations of any kind.

[7] `https://github.com/bheisler/criterion.rs`, Accessed 2023-01-25. Source code of linear interpolation for computing quantiles: `https://github.com/bheisler/criterion.rs/blob/27642b476837753cbb539f269fbbcbefa815bf00/src/stats/univariate/percentiles.rs#L20`

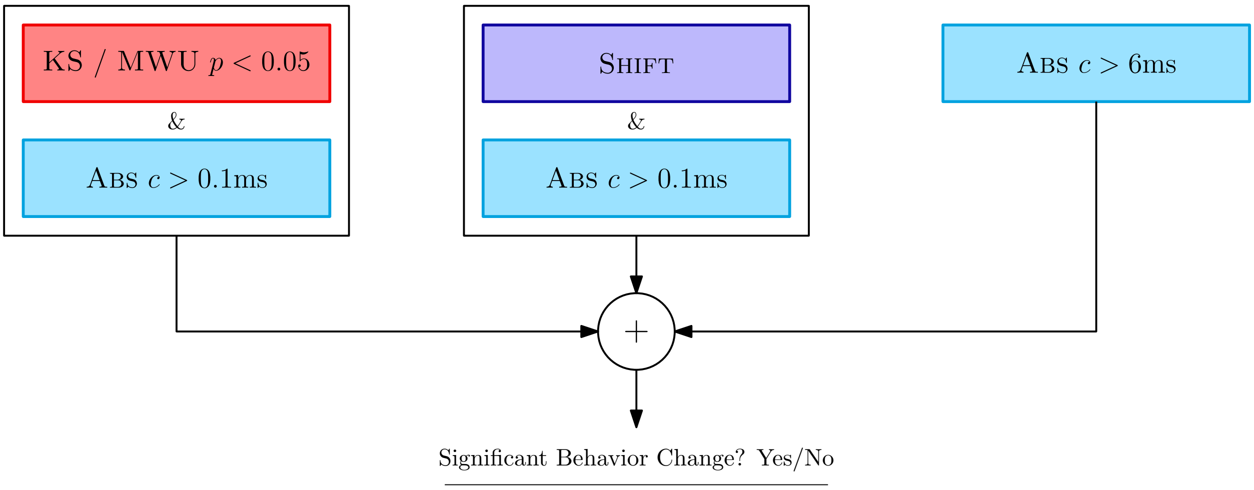


Figure 3.9: Flowchart of our combined detection method. "&" represents an "and"-combination, where both detection methods must give positive results for the input to be deemed significantly different. "+" represents an "or"-combination. Our different detection methods are implemented such that it is easy to interchange and combine them.

3.6 Visualizations

In Section 2.3.2, we describe how MIDS visualizes differences between system models on six different levels of abstraction. In MIDS+TIME, we extend MIDS' views with timing-related visualizations where suitable, in particular the Level 6 which shows service fragment difference automata, and add a new "Timing Overview" view. The latter is intended as an overview for the most important timing data of the two examined traces.

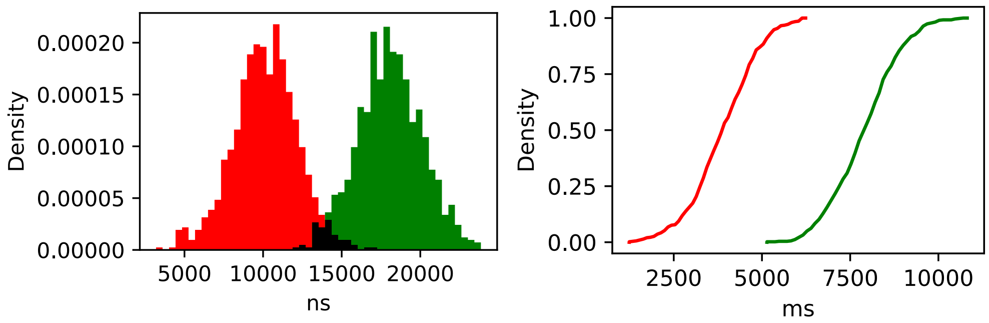
Before showing the full views, let us briefly introduce the different plot types which are suitable for comparing pairs of time samples.

3.6.1 Histograms and Cumulative Density Plots

Histograms are an obvious choice for visualizing sample distributions. We overlay the two (normalized) histograms of T_L, T_R and color overlapping regions black, while areas uniquely covered by T_L or T_R will be colored red or green, respectively. This matches the coloring of the difference automata and again intends to mimick the intuitive coloring of a "`git diff`" output. Values which have decreased in frequency (i.e., are more common in T_L than in T_R) will appear red; values which have increased in frequency (i.e., are more common in T_R than in T_L) appear green.

A histogram as shown in Figure 3.10a may thus indicate that a measured functionality got slower, because the smaller values to the left are red, i.e., only present in the reference distribution, and seem to have moved to the right in the current distribution.

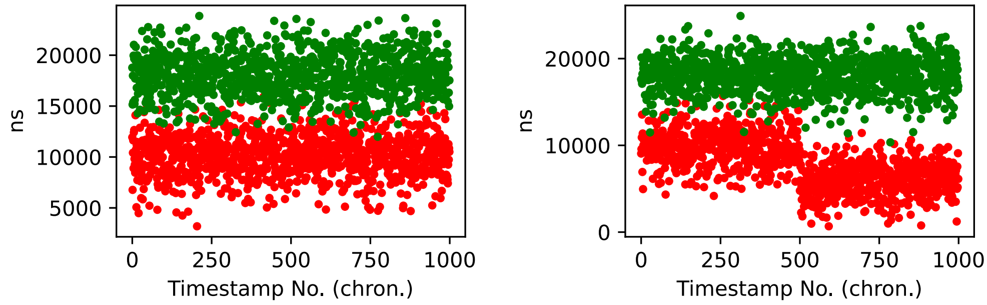
We can also visualize the empirical cumulative distribution functions (CDF) F_L, F_R of the two time samples, shown in Figure 3.10b. While not quite as intuitive as plain histograms, one advantage of the CDF visualization is that it does not depend on the choice of a bucket size, like histograms do.



(a) Example of two overlapped histograms.

(b) CDFs for the samples shown in (a).

Figure 3.10: Histogram and CDF function examples.



(a) No behavior change at runtime.

(b) Behavior change after half the runtime.

Figure 3.11: Examples of control charts.

3.6.2 Control Charts

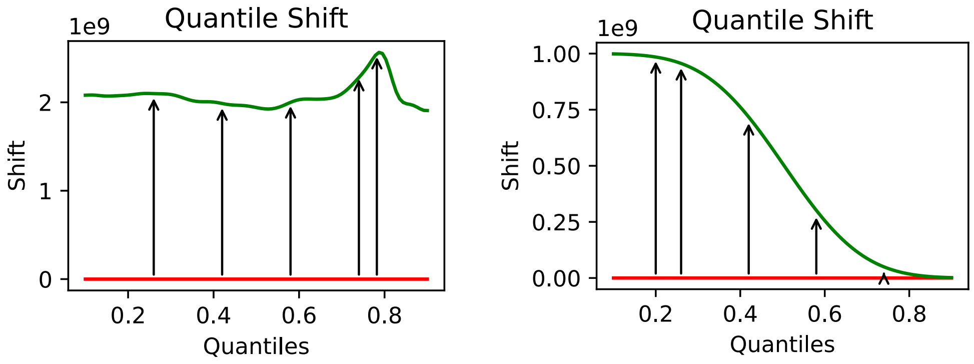
Histograms, while useful, do not capture the temporal evolution of taken measurements. Inspired by the concept of control charts (cf. [28,36]), we plot the collected time samples in chronological order. Figure 3.11a shows the control chart corresponding to the histogram in Figure 3.10a; Figure 3.11b shows a control chart where the measured functionality seems to show a speed-up after about half the observed system runtime. It has to be noted that the horizontal space between sample points is constant and does not represent the elapsed time between the occurence of the measurements.

3.6.3 Shift Functions

Shift functions, which were introduced by Kjell Doksum, visualize the difference between two samples [12]. They do so by plotting the difference between matching quantile values of the two distributions, i.e.:

$$\text{shift}(x) = Q_{R,x} - Q_{L,x}$$

We again use Harrel-Davis quantile estimates [15] and again cut off 10% of the lowest and highest quantiles for stability reasons. For better visualization, we draw a reference zero line in red, and the shift function in green (cf. Figure 3.12a). The green function lying above the red line indicates that the quantiles were shifted positively, i.e., the timing values have increased. The



(a) Positive shift of all quantiles. (b) Positive shift of the lower quantiles.

Figure 3.12: Examples of shift functions. On the left, we see a shift function that is above the zero-line at all times, which indicates a positive shift (by $\sim 2\cdot10^9$) of all quantiles. On the right, we can see that only the lower quantiles have been shifted, meaning that only smaller time values have increased, while larger time values have not changed much. This could for example be explained by a bimodal data distribution, where only the left peak has been shifted in the changed trace.

green lying below the red indicates that the timing values have decreased. We also add arrows to the plot to emphasize that a shift has occured. The shift function works well for multi-modal distributions: When only one mode of a distribution was shifted, this is visible in the shift function because only part of the quantiles will have changed, while other quantiles may still lie unchanged on the reference line [12, 34]. An example for this is shown in Figure 3.12b.

Visually, our Shift detection method classifies a pair of samples as significantly different if the shift function is entirely greater or equal to zero (resp. less than or equal to zero) at the sampled points.

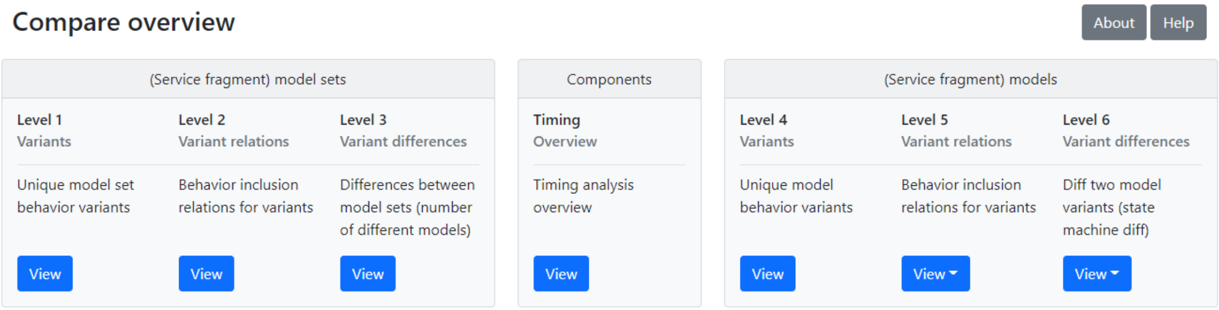
3.6.4 Mids+Time User Interface

In this section, we describe how all the building blocks introduced in the previous sections – time-annotated difference automata, detection heuristics, visualizations – are combined into one user interface in Mids+Time. Because we can not show ASML-internal data, we instrument the open-source tool cURL[8] to produce traces in the same format as the company's software[9]. We then use the instrumented cURL version to make requests to `https://www.google.com`, once using a WiFi-connection, and once using a cellular connection, thus obtaining traces with different timings. Because cURL on its own is not component-based software in the classical sense, we define different compilation units to be different components, function calls between compilation units as communication, and functions as service fragments. The top most called function in Figure 3.14, is displayed in Mids as "`lib_cookie_c.lib_cookie_h_ICurl__Curl_cookie_freelist__blk_blk_lib_cookie_c`". This corresponds to the function `Curl_cookie_freelist` which is defined in `lib/cookie.c`[10]. Here, `lib/cookie.c` is considered the component for the service `Curl_cookie_freelist`. All

[8] `https://curl.se/`, Accessed 2022-01-16.

[9] Details on the instrumentation can be found in Appendix A.1.

[10] `https://github.com/curl/curl/blob/11708d6f006db0960c8ff6eab2fd9b1d15d7e178/lib/cookie.c#L1553`



Compare overview

About Help

(Service fragment) model sets

Level 1
Variants
Unique model set behavior variants
View

Level 2
Variant relations
Behavior inclusion relations for variants
View

Level 3
Variant differences
Differences between model sets (number of different models)
View

Components

Timing
Overview
Timing analysis overview
View

(Service fragment) models

Level 4
Variants
Unique model behavior variants
View

Level 5
Variant relations
Behavior inclusion relations for variants
View

Level 6
Variant differences
Diff two model variants (state machine diff)
View

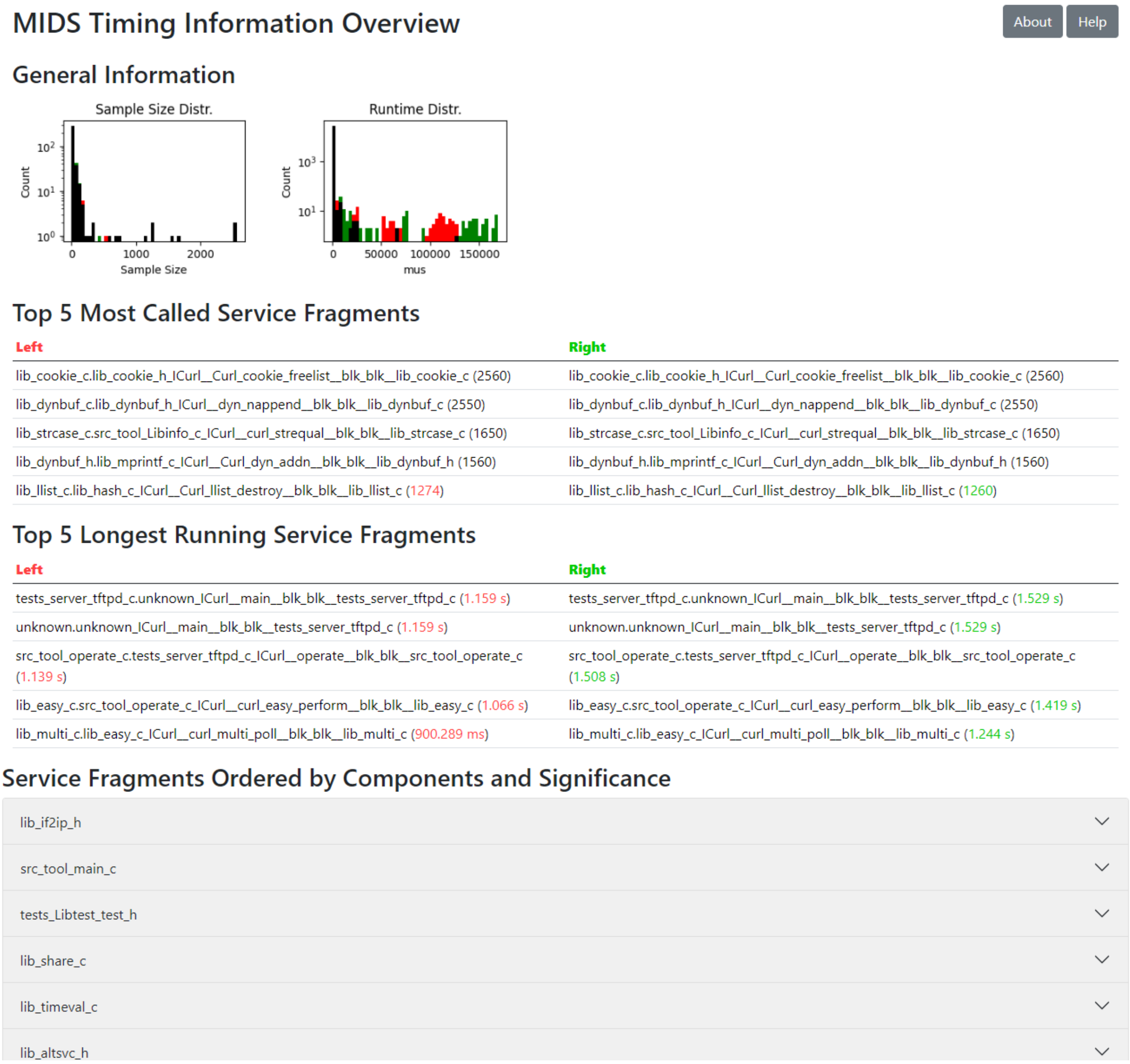
Figure 3.13: Mids+Time main menu.

function calls are naively considered blocking calls (hence the "`blk`"; see also Section 2.4) as is done for ASML traces without information about the used communication patterns.

Our analysis of the cURL traces using Mids+Time show various increases in function execution times, in particular in components related to executing network requests and polling, which reflect the expected increase in network response time due to the cellular connection being slower than the WiFi-connection.

The output of Mids is generated in the form of an interactive HTML page, where the user can separately access the six levels of granularity (cf. Section 2.3.2) from a main menu (Figure 3.13). For Mids+Time, we have added a seventh view, the timing overview page (shown in Figure 3.14).It contains some general information on top: Histograms for the service fragment call count distribution and the service fragment runtime distribution, as well as a list of the top 5 most called service fragments and the top 5 longest running service fragments. These are intended to give the user an immediate overview of any major changes. For our cURL example traces, we can see that the runtimes in the changed trace (i.e., the trace using cellular communication), many service fragments have increased runtime. This intuitively makes sense, as we would expect cellular communication to happen at a slower pace, affecting the runtime of many functions. Changes in the top 5 lists are highlighted in the usual red and green color scheme. Below this, a list of every system component is shown. It can be expanded to reveal a sublist of every service fragment within that component, as shown in Figure 3.15. These sublists are sorted by the magnitude of their runtime changes: Service fragments where the runtime between the two input traces has varied the most, will be shown on top. If the runtime samples are deemed to constitute significant differences, the service fragment's name will be rendered in purple. This can also be seen in Figure 3.15: In our cURL example, it seems that the function `curl_multi_poll` takes over 340 ms longer in the trace using cellular communication. Every item in the service fragment list contains a small preview of the service fragment's difference automaton, as well as a histogram, control chart, and shift plot of the runtime data. Clicking on a service fragment reveals a table with the sample size (i.e., call frequency), minimum, 25th-percentile, median, 75th-percentile and maximum runtime for the left and right traces, respectively. If the runtime time samples of a service fragment are detected as significantly different by our heuristics, the list entry will be highlighted with a purple border and a purple heading.

Clicking on the preview of the difference automata brings the user to the level 6 view of the



MIDS Timing Information Overview

General Information

Top 5 Most Called Service Fragments

Left	Right
lib_cookie_c.lib_cookie_h_ICurl__Curl_cookie_freelist__blk_blk__lib_cookie_c (2560)	lib_cookie_c.lib_cookie_h_ICurl__Curl_cookie_freelist__blk_blk__lib_cookie_c (2560)
lib_dynbuf_c.lib_dynbuf_h_ICurl__dyn_nappend__blk_blk__lib_dynbuf_c (2550)	lib_dynbuf_c.lib_dynbuf_h_ICurl__dyn_nappend__blk_blk__lib_dynbuf_c (2550)
lib_strcase_c.src_tool_Libinfo_c_ICurl__curl_strequal__blk_blk__lib_strcase_c (1650)	lib_strcase_c.src_tool_Libinfo_c_ICurl__curl_strequal__blk_blk__lib_strcase_c (1650)
lib_dynbuf_h.lib_mprintf_c_ICurl__Curl_dyn_addn__blk_blk__lib_dynbuf_h (1560)	lib_dynbuf_h.lib_mprintf_c_ICurl__Curl_dyn_addn__blk_blk__lib_dynbuf_h (1560)
lib_llist_c.lib_hash_c_ICurl__Curl_llist_destroy__blk_blk__lib_llist_c (1274)	lib_llist_c.lib_hash_c_ICurl__Curl_llist_destroy__blk_blk__lib_llist_c (1260)

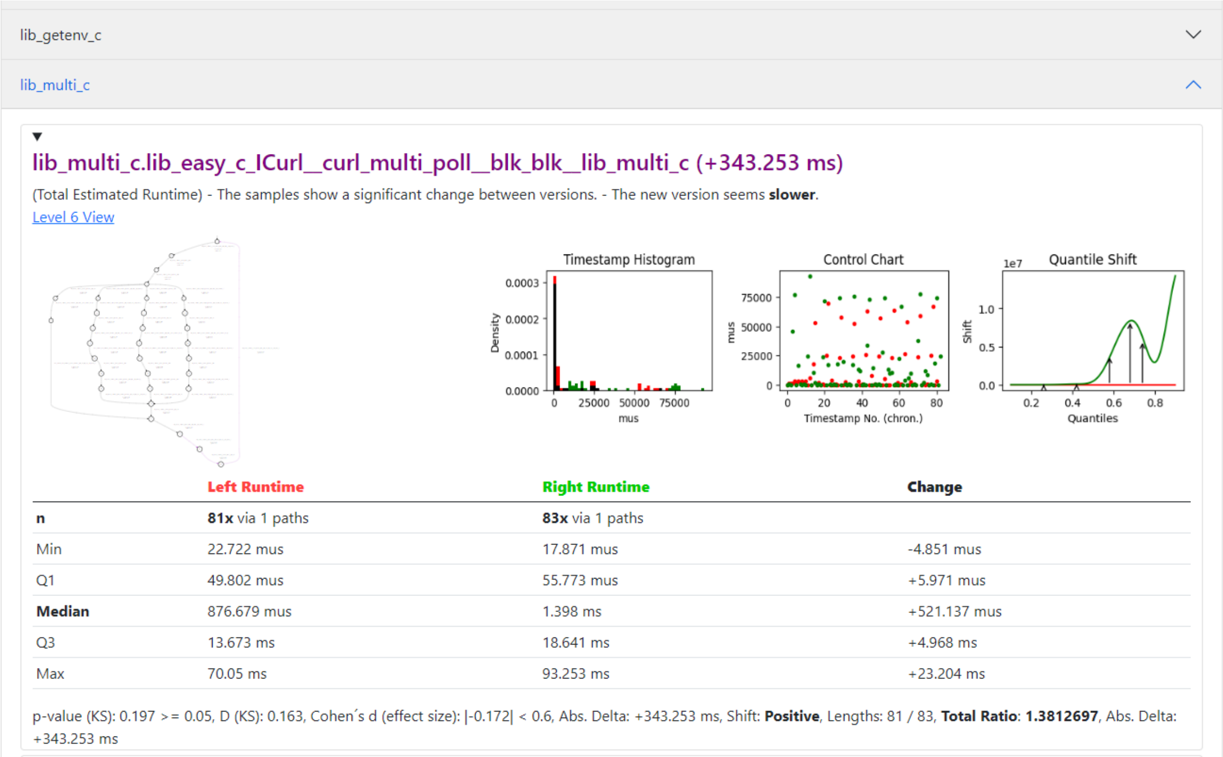
Top 5 Longest Running Service Fragments

Left	Right
tests_server_tftpd_c.unknown_ICurl__main__blk_blk__tests_server_tftpd_c (1.159 s)	tests_server_tftpd_c.unknown_ICurl__main__blk_blk__tests_server_tftpd_c (1.529 s)
unknown.unknown_ICurl__main__blk_blk__tests_server_tftpd_c (1.159 s)	unknown.unknown_ICurl__main__blk_blk__tests_server_tftpd_c (1.529 s)
src_tool_operate_c.tests_server_tftpd_c_ICurl__operate__blk_blk__src_tool_operate_c (1.139 s)	src_tool_operate_c.tests_server_tftpd_c_ICurl__operate__blk_blk__src_tool_operate_c (1.508 s)
lib_easy_c.src_tool_operate_c_ICurl__curl_easy_perform__blk_blk__lib_easy_c (1.066 s)	lib_easy_c.src_tool_operate_c_ICurl__curl_easy_perform__blk_blk__lib_easy_c (1.419 s)
lib_multi_c.lib_easy_c_ICurl__curl_multi_poll__blk_blk__lib_multi_c (900.289 ms)	lib_multi_c.lib_easy_c_ICurl__curl_multi_poll__blk_blk__lib_multi_c (1.244 s)

Service Fragments Ordered by Components and Significance

- lib_if2ip_h
- src_tool_main_c
- tests_Libtest_test_h
- lib_share_c
- lib_timeval_c
- lib_altsvc_h

Figure 3.14: Mids+Time timing data overview with general data on top and expandable list of components on the bottom.



lib_getenv_c

lib_multi_c

▼ lib_multi_c.lib_easy_c_lCurl__curl_multi_poll__blk_blk__lib_multi_c (+343.253 ms)

(Total Estimated Runtime) - The samples show a significant change between versions. - The new version seems **slower**.

Level 6 View

	Left Runtime	Right Runtime	Change
n	**81x** via 1 paths	**83x** via 1 paths	
Min	22.722 mus	17.871 mus	-4.851 mus
Q1	49.802 mus	55.773 mus	+5.971 mus
Median	876.679 mus	1.398 ms	+521.137 mus
Q3	13.673 ms	18.641 ms	+4.968 ms
Max	70.05 ms	93.253 ms	+23.204 ms

p-value (KS): 0.197 >= 0.05, D (KS): 0.163, Cohen´s d (effect size): |-0.172| < 0.6, Abs. Delta: +343.253 ms, Shift: **Positive**, Lengths: 81 / 83, **Total Ratio**: **1.3812697**, Abs. Delta: +343.253 ms

Figure 3.15: Expanding a component in the timing overview presents a list of service fragments, ordered by the magnitude of their timing differences, if any are present. Here, a service fragment with an execution time change of approximately +343 ms is listed first in component “`lib_multi_c`”.

service fragment. This can also be accessed through the main menu. An example is shown in Figure 3.16. We have extended Mids’ difference automata concept in several ways. First, transitions are annotated with their frequency (i.e., number of occurences) in each trace, and a small inline histogram plot of their time samples. Second, transitions with automatically detected significant differences are highlighted in purple, as we also do in Figures 3.3 and 3.4. This intuitively extends the difference automata’s color palette of red (“removed behavior”), green (“added behavior”), black (“unchanged”) with purple (“changed timing”). Third, clicking on a transition brings up a side panel with more in-depth timing information. It contains the same table as also used for the service fragment runtimes, but with the transition time samples T_L, T_R. This allows quickly seeing the minimum, median and maximum time values. For computation of the quartiles, we again utilize the Harrell-Davis quantile estimator [15]. Below the table, a histogram, control chart, cumulative density plot, and a shift plot of the time samples are shown. If the transition has been detected to contain significant timing changes, a purple box will be shown in the side panel with information on how this conclusion was reached (e.g., p-value, absolute difference).

Lastly, the level 4 view, which is accessible through the main menu, has also been extended (see Figure 3.17). If the input traces contain communication metadata, we display our Communication Pattern metrics for each service fragment in a table-based view. Figure 3.17 shows an excerpt of the table-based view for a pair of real-world traces. There are three columns for each statistic: The left shows the total time for the reference trace, the right for the changed trace. They are color-coded to show service fragments with higher values in red and lower values and green, to

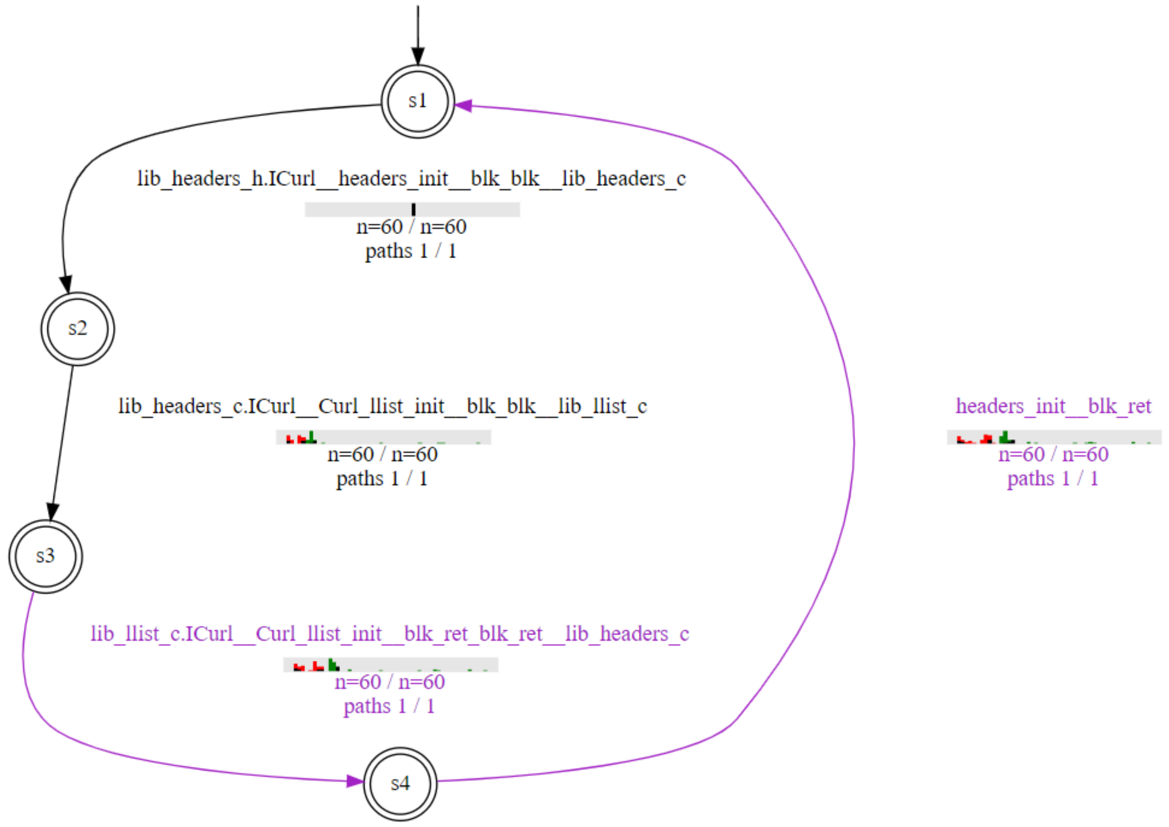
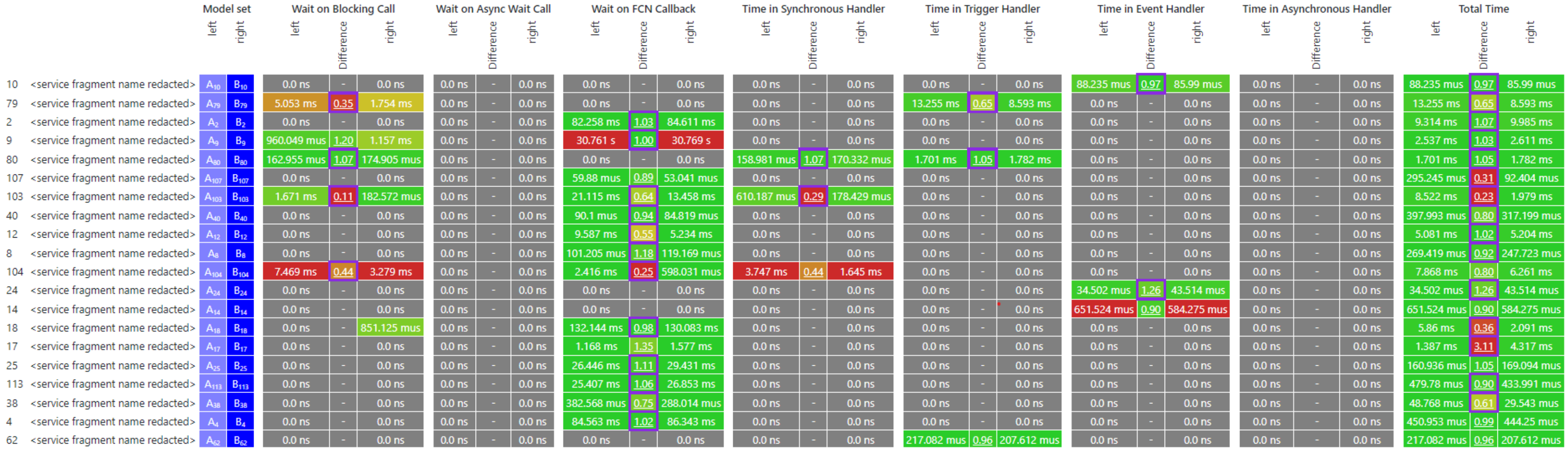

s1
lib_headers_h.ICurl__headers_init__blk_blk__lib_headers_c
n=60 / n=60
paths 1 / 1
s2
lib_headers_c.ICurl__Curl_llist_init__blk_blk__lib_llist_c
n=60 / n=60
paths 1 / 1
s3
lib_llist_c.ICurl__Curl_llist_init__blk_ret_blk_ret__lib_headers_c
n=60 / n=60
paths 1 / 1
s4
headers_init__blk_ret
n=60 / n=60
paths 1 / 1

Figure 3.16: Example difference automata from the cURL traces. The right transition label has been shortened for presentation purposes. The purple transitions indicate a timing change, and this is also visible in the small inline histograms below the transition labels: The majority of the red and green distribution's masses move apart visibly.



		Model set		Wait on Blocking Call			Wait on Async Wait Call			Wait on FCN Callback			Time in Synchronous Handler			Time in Trigger Handler			Time in Event Handler			Time in Asynchronous Handler			Total Time		
		left	right	left	Difference	right	left	Difference	right	left	Difference	right	left	Difference	right	left	Difference	right	left	Difference	right	left	Difference	right	left	Difference	right
10	<service fragment name redacted>	A_{10}	B_{10}	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	88.235 mus	0.97	85.99 mus	0.0 ns	-	0.0 ns	88.235 mus	0.97	85.99 mus
79	<service fragment name redacted>	A_{79}	B_{79}	5.053 ms	0.35	1.754 ms	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	13.255 ms	0.65	8.593 ms	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	13.255 ms	0.65	8.593 ms
2	<service fragment name redacted>	A_2	B_2	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	82.258 ms	1.03	84.611 ms	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	9.314 ms	1.07	9.985 ms
9	<service fragment name redacted>	A_9	B_9	960.049 mus	1.20	1.157 ms	0.0 ns	-	0.0 ns	30.761 s	1.00	30.769 s	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	2.537 ms	1.03	2.611 ms
80	<service fragment name redacted>	A_{80}	B_{80}	162.955 mus	1.07	174.905 mus	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	158.981 mus	1.07	170.332 mus	1.701 ms	1.05	1.782 ms	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	1.701 ms	1.05	1.782 ms
107	<service fragment name redacted>	A_{107}	B_{107}	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	59.88 mus	0.89	53.041 mus	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	295.245 mus	0.31	92.404 mus
103	<service fragment name redacted>	A_{103}	B_{103}	1.671 ms	0.11	182.572 mus	0.0 ns	-	0.0 ns	21.115 ms	0.64	13.458 ms	610.187 mus	0.29	178.429 mus	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	8.522 ms	0.23	1.979 ms
40	<service fragment name redacted>	A_{40}	B_{40}	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	90.1 mus	0.94	84.819 mus	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	397.993 mus	0.80	317.199 mus
12	<service fragment name redacted>	A_{12}	B_{12}	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	9.587 ms	0.55	5.234 ms	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	5.081 ms	1.02	5.204 ms
8	<service fragment name redacted>	A_8	B_8	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	101.205 mus	1.18	119.169 mus	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	269.419 mus	0.92	247.723 mus
104	<service fragment name redacted>	A_{104}	B_{104}	7.469 ms	0.44	3.279 ms	0.0 ns	-	0.0 ns	2.416 ms	0.25	598.031 mus	3.747 ms	0.44	1.645 ms	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	7.868 ms	0.80	6.261 ms
24	<service fragment name redacted>	A_{24}	B_{24}	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	34.502 mus	1.26	43.514 mus	0.0 ns	-	0.0 ns	34.502 mus	1.26	43.514 mus
14	<service fragment name redacted>	A_{14}	B_{14}	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	651.524 mus	0.90	584.275 mus	0.0 ns	-	0.0 ns	651.524 mus	0.90	584.275 mus
18	<service fragment name redacted>	A_{18}	B_{18}	0.0 ns	-	851.125 mus	0.0 ns	-	0.0 ns	132.144 ms	0.98	130.083 ms	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	5.86 ms	0.36	2.091 ms
17	<service fragment name redacted>	A_{17}	B_{17}	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	1.168 ms	1.35	1.577 ms	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	1.387 ms	3.11	4.317 ms
25	<service fragment name redacted>	A_{25}	B_{25}	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	26.446 ms	1.11	29.431 ms	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	160.936 mus	1.05	169.094 mus
113	<service fragment name redacted>	A_{113}	B_{113}	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	25.407 ms	1.06	26.853 ms	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	479.78 mus	0.90	433.991 mus
38	<service fragment name redacted>	A_{38}	B_{38}	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	382.568 mus	0.75	288.014 mus	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	48.768 mus	0.61	29.543 mus
4	<service fragment name redacted>	A_4	B_4	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	84.563 ms	1.02	86.343 ms	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	450.953 mus	0.99	444.25 mus
62	<service fragment name redacted>	A_{62}	B_{62}	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	217.082 mus	0.96	207.612 mus	0.0 ns	-	0.0 ns	0.0 ns	-	0.0 ns	217.082 mus	0.96	207.612 mus

Figure 3.17: Level 4 view, extended with different timing metrics.

help more easily spotting outliers, i.e., service fragments with particularly high or low durations in one category. The center columns show the ratio between the left and right values. Clicking on the center cell displays a control chart of the time samples. If the time samples are detected as significantly different by our heuristics, the center cell will be marked with a purple border.

4 Evaluation

As stated in Chapter 3, the definition of a timing regression is fluid and subjective; In particular, a timing change may be problematic in some scenarios, but irrelevant or disregarded as noise in others. The combined detection method which is used in our tool is tuned to our subjective notion of what faults are significant, based on feedback from the engineers. This makes it difficult to evaluate the performance of our detection method objectively.

In the following, we first show that our detection methods work in principle, using simple, handcrafted example traces. We then further evaluate them by injecting randomized artificial faults into real-world program traces. The goal of these two first evaluations using artificial traces is to show that the individual parts of our combined detection method are **complementary**, i.e., they detect distinct sets of fault types, and **effective**, meaning they achieve reasonable recall and precision. Finally and most importantly, we evaluate Mids+Time on real-world traces from ASML systems, reporting on found timing behavior changes and according feedback by the responsible engineers.

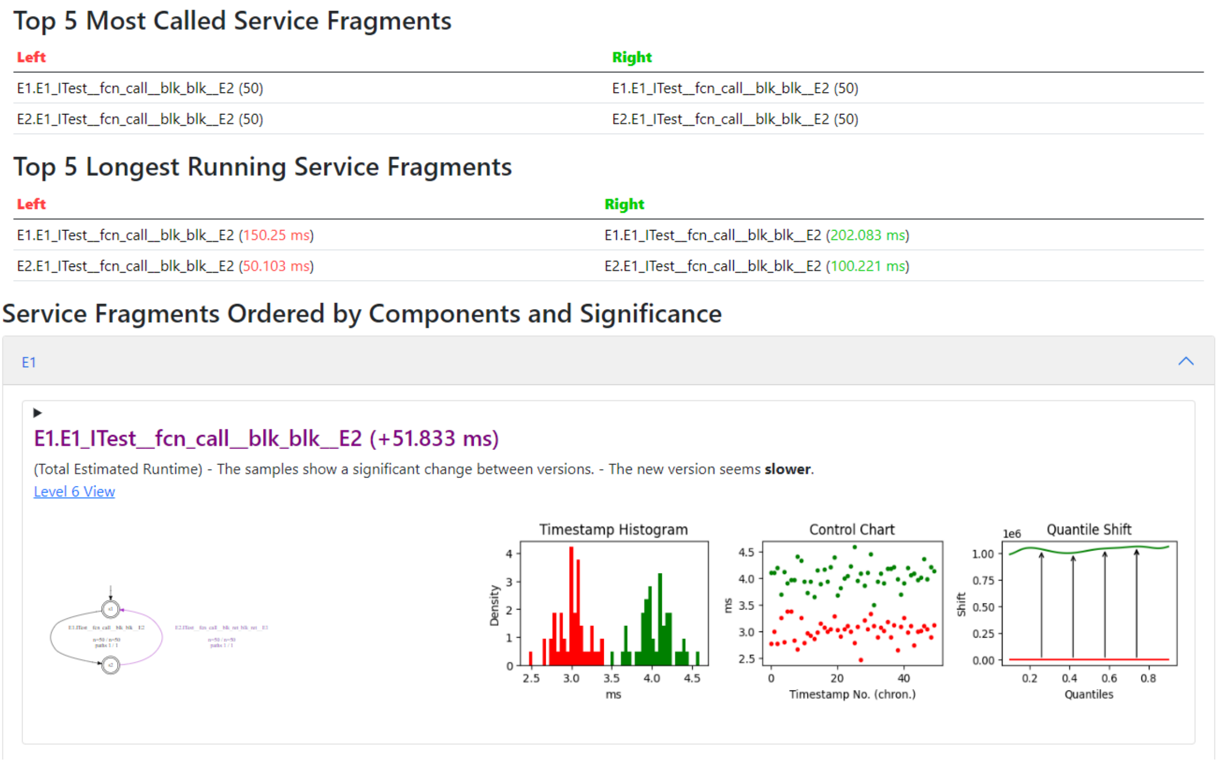
4.1 Simple Timing Behavior Change Tests

We start with small custom traces, which were generated by us specifically to test examples of simple timing behavior change scenarios. As a reference trace, we use a simple trace with just four repeating events, as shown in Figure 4.1. It consists of a component “`E1`” making a blocking call to component “`E2`”. When referring to the “function runtime” in the following, we mean the time between the second and third event, i.e., the time between Entry and Exit event of the function on component `E2`. All events occur one millisecond apart. We add a small error $\varepsilon \sim \mathcal{N}(0, 10^{-4})$ to each event time, to simulate noise with a variance of 0.1 ms.

For every test case, we now describe the change and the result of the analysis performed by Mids+Time. Because we have already discussed how Mids+Time visualizes the analysis results in the last chapter, we do not describe the analysis output in detail.

```
0.000 E1 > blk_call !m1
0.001 E2 > blk_call ?m1
0.002 E2 < blk_call !ret1
0.003 E1 < blk_call ?ret1
 ... (repeated 50 times) ...
```

Figure 4.1: A minimal generated trace which was used as reference trace in most of our test cases. The timestamps are given in seconds.



Top 5 Most Called Service Fragments

Left	Right
E1.E1_ITest__fcn_call__blk_blk__E2 (50)	E1.E1_ITest__fcn_call__blk_blk__E2 (50)
E2.E1_ITest__fcn_call__blk_blk__E2 (50)	E2.E1_ITest__fcn_call__blk_blk__E2 (50)

Top 5 Longest Running Service Fragments

Left	Right
E1.E1_ITest__fcn_call__blk_blk__E2 (150.25 ms)	E1.E1_ITest__fcn_call__blk_blk__E2 (202.083 ms)
E2.E1_ITest__fcn_call__blk_blk__E2 (50.103 ms)	E2.E1_ITest__fcn_call__blk_blk__E2 (100.221 ms)

Service Fragments Ordered by Components and Significance

E1

E1.E1_ITest__fcn_call__blk_blk__E2 (+51.833 ms)

(Total Estimated Runtime) - The samples show a significant change between versions. - The new version seems **slower**.

Level 6 View

Figure 4.2: Detail of MIDS+TIME's time overview page for test 1.

The following simple tests were conducted:

1. **Test:** Increased function runtime.

   Description: We double the runtime of the called function on component E2.

   Result: A timing behavior change is correctly reported on the exit edges of both component's service fragments. Figure 4.2 shows an excerpt of MIDS+TIME's time overview page for this test case.

2. **Test:** Small outlier.

   Description: In one of the fifty repetitions, we double the runtime of the called function (for a total runtime of 2 ms).

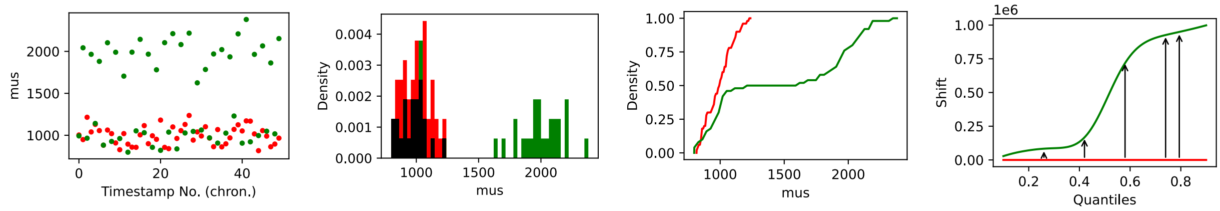
   Result: No timing behavior change is reported. This is expected, as the change is relatively small. The ABS detection mechanism's threshold is not exceeded, and the change does not alter the distribution's shape significantly.

3. **Test:** Large outlier.

   Description: In one of the fifty repetitions, we increase the runtime of the called function by 10 ms.

   Result: A timing behavior change is reported, due to the ABS detection mechanism's threshold being exceeded.

4. **Test:** Unimodal to bimodal change (Half the calls slower).



(a) Control chart.

(b) Histogram.

(c) CDFs.

(d) Shift function.

Figure 4.3: Time sample plots for the Exit of `blk_call` on `E2` in test 5.

Description: Half the function's runtimes are doubled, which results in a bimodal time sample distribution in the changed trace (cf. Figure 4.3).

Result: The timing behavior change is correctly reported.

5. **Test:** Bimodal to bimodal change.

   Description: The reference trace's function runtime now follows a bimodal distribution with means 1 ms and 2 ms, simulating a function which is called with different parameters. In the changed trace, the larger mean is changed to 4 ms.

   Result: The timing behavior change is correctly reported.

6. **Test:** Higher call frequency.

   Description: In the reference trace, the function runtime is set to 3 ms. In the changed trace, it is set to 2 ms, but the function is called twice as often (i.e., the trace contains twice as many events).

   Result: The timing behavior change is correctly reported.

During development, we implemented further tests which also included functional changes as well as different communication patterns, however the tests listed here should already be sufficient to show that our method is able to handle simple scenarios. They also highlight how our detection methods are complementary: While the large single outlier of test 3 is detected with Abs, it is not detected by KS. In turn, KS is able to detect more complicated changes in the shape of timing distributions (tests 4-6), which could be missed by Abs if their absolute magnitude were smaller.

The presented small test traces show that our combined detection method can handle simple timing behavior change scenarios.

4.2 Artificial Fault Injection

While we have now looked at artificially created timing behavior changes, these manually created traces were fairly small and had limited complexity. They also contained strict assumptions about the kinds of timing behavior changes we expect to encounter and how they influence the traces. In this next experiment, we want to reduce the number of assumptions and increase the trace's complexity by injecting different types of randomized timing changes (*faults*) into a real-world system trace.

In the following, we describe how the fault injection is performed, what kinds of faults we are injecting, how we evaluate the detection algorithms' performances, and finally present and discuss the results.

4.2.1 Fault Injection Algorithm

A *fault* is a change of timing of one or more events in a trace. We formally define a single fault as a tuple $(e, \delta) \in E \times \mathbb{R}$, where e is the faulty event out of a set of events E, and δ is the amount of time by which it is moved – a positive sign indicates that e occurs later, a negative sign indicates that e occurs earlier. For the purpose of this evaluation, we ignore functional changes, meaning we want to shift events while **preserving the order** of dependent events.

To achieve this, a trace is modeled as a directed acyclic graph, where the nodes E are the events and edges between events constrain the order of events: If an edge (e_1, e_2) exists, event e_2 must happen after e_1. $t(e)$ denotes the time at which e occurs. We define a slack function $s : E \times E \to \mathbb{R}$, which assigns to each edge the slack time between the adjacent events. $s(e_1, e_2) = 10$ indicates that e_2 can occur up to 10 time units earlier than it currently does without violating any constraints. We model all message dependencies (e.g., calls between components) to have a slack time of zero, to prevent any communication happening faster in the faulty trace than in the reference trace. We model independent events on the same lifeline to have a slack time of larger than zero, allowing them to move closer together without changing order.

Given a reference trace with event times t_R and slack function s_R, a changed trace with event times t, and two events e_1, e_2 which are connected by (e_1, e_2),

$$t(e_2) - t(e_1) \geq t_R(e_2) - t_R(e_1) - s_R(e_1, e_2) \tag{4.2.1}$$

must hold for the changed trace to be *consistent* with the reference trace. Put in words, the events in the changed trace may not happen quicker than in the reference trace, save for the allowed slack. Note that the definition of the slack function allows for negative values; During the execution of our algorithm, an edge may temporarily have negative slack. Negative slack indicates that events must move further apart, or otherwise the trace will be inconsistent with the reference trace. However, for any trace that is an input to our algorithm, and any trace that is output by the algorithm, the slack on an edge should not be negative or exceed the time between the two adjacent events:

$$\forall \text{ edges } (e_1, e_2) : \quad 0 \leq s(e_1, e_2) \leq t(e_2) - t(e_1) \tag{4.2.2}$$

Algorithm 2 is a simple algorithm to inject one fault (e, δ) into a trace while keeping it consistent. It is based on propagating the fault in a breadth-first manner in the direction of the fault. For simplicity, consider a fault with $\delta > 0$. This fault will only affect events to its right. We start with the original faulty event and add *shift* $= \delta$ to its start time and update the slack values on its outgoing edges accordingly ("updateSlack"). Then, we recursively add all the event's successors

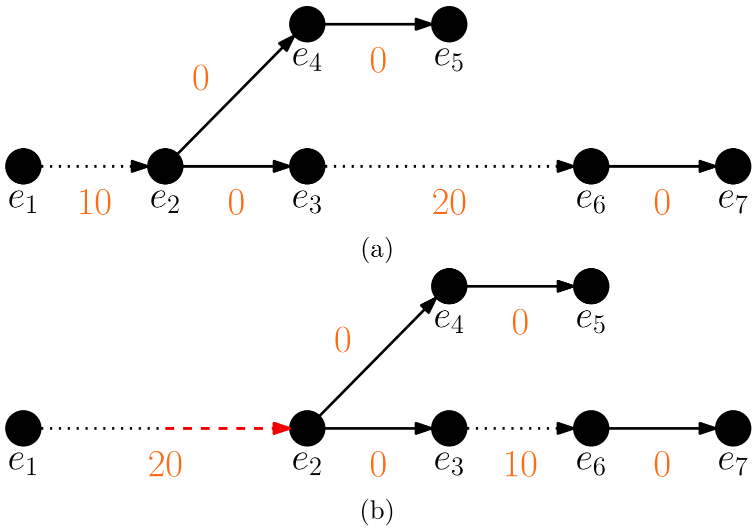


Figure 4.4: Example schedule (a) before and (b) after applying fault $(e_2, +10)$. Orange numbers denote edge slack time. Edges with non-zero slack are drawn as dotted arrows for visualization purposes only. The red arrow symbolizes the original injected fault $(e_2, +10)$.

to the queue. For each successive event in the queue, we determine the amount it is required to *shift* to the right by determining the minimum remaining slack time at any of its incoming edges (“computeRequiredShift”), and subtracting that from its start time. If the slack time on all incoming edges is positive or zero, no shift is required. For faults with $\delta < 0$, the steps are analogous, but now propagating from right to left.

Figure 4.4 shows an example schedule before and after injecting a fault $(e_2, +10)$ with Algorithm 2. Note how edge (e_3, e_6) can absorb the incurred delay because it has sufficient slack.

It is easy to see that the algorithm terminates, as it propagates faults strictly in one direction and in the process only ever adds *either* successors *or* predecessors of an event to the queue. At one point, the beginning or the end of the trace will be reached, and the queue will drain until it is empty. We do not provide a detailed proof that the algorithm is correct, but let us give an intuitive argument for $\delta > 0$ by showing that Equation (4.2.1) holds after using the algorithm to inject a fault $(e_1, +\delta)$ into a reference trace with $e_1, e_2 \in E$ and edge (e_1, e_2).

The algorithm will first shift e_1 by δ and update the outgoing edge’s slack:

$$t(e_1) = t_R(e_1) + \delta$$
$$s(e_1, e_2) = s_R(e_1, e_2) - \delta$$

Algorithm 2 Fault Injection – Breadth-First Propagation

```
Input: Fault $(e, \pm\delta) \in E \times \mathbb{R}$
Input: Slack Function $s : E \times E \to \mathbb{R}$
Output: Consistent schedule with injected fault $(e, \pm\delta)$.
  procedure Main
      $Q \leftarrow [e]$                                          ▷ $Q$ is a simple queue.
      while $Q \neq [\,]$ do
          $current \leftarrow \mathrm{pop}(Q)$

          if $current = e$ then
              $shift \leftarrow \delta$                                 ▷ This is the faulty event.
          else
              $shift \leftarrow$ computeRequiredShift($s$, $current$, $\delta$)
          end if

          if $shift = 0$ then
              continue while                                    ▷ No changes need to be made.
          else if $shift > 0$ then
              for $e_s \in \mathrm{succ}(current)$ do                     ▷ Propagate to the right.
                  $\mathrm{push}(Q, e_s)$
              end for
          else
              for $e_p \in \mathrm{pred}(current)$ do                     ▷ Propagate to the left.
                  $\mathrm{push}(Q, e_p)$
              end for
          end if

          applyShift($current$, $shift$)                           ▷ Adjust event time
          updateSlack(s, $current$, $shift$)
      end while
  end procedure

  procedure computeRequiredShift($s, e, \delta$)
      if $\delta > 0$ then
          return $-\min\left\{ \min_{e_p \in \mathrm{pred}(e)} s(e_p, e),\ 0 \right\}$   ▷ Shift right by most negative incoming slack.
      else
          return $\min\left\{ \min_{e_s \in \mathrm{succ}(e)} s(e, e_s),\ 0 \right\}$    ▷ Shift left by most negative outgoing slack.
      end if
  end procedure

  procedure updateSlack($s, e, \delta$)                       ▷ Update slack on incident edges.
      for $e_s \in \mathrm{succ}(e)$ do
          $s(e, e_s) \leftarrow s(e, e_s) - \delta$
      end for
      for $e_p \in \mathrm{pred}(e)$ do
          $s(e_p, e) \leftarrow s(e_p, e) + \delta$
      end for
  end procedure
```

The shift for e_2 will be calculated by computeRequiredShift:

$$\begin{aligned}
t(e_2) = t_R(e_2) - \min\left\{\min_{e_p \in \text{pred}(e)} s(e_p, e_2),\ 0\right\} \\
= t_R(e_2) - \min\left\{\min\left\{s(e_1, e_2), ...\right\},\ 0\right\} \\
= t_R(e_2) - \min\left\{\min\left\{s_R(e_1, e_2) - \delta, ...\right\},\ 0\right\} \\
\geq t_R(e_2) - \begin{cases} 0 & \text{if } \delta \leq s_R(e_1, e_2) \\ (s_R(e_1, e_2) - \delta), & \text{otherwise} \end{cases} =: \mathsf{LB}
\end{aligned}$$

Note that the last expression is a lower bound, because other incoming edges (denoted by the ellipsis) may have a more negative slack value. We continue with this lower bound. Let us first consider the case where $\delta \leq s_R(e_1, e_2)$ and replace $t(e_1), t(e_2)$ in Equation (4.2.1):

$$\underbrace{t_R(e_2)}_{\mathsf{LB}} - t_R(e_1) - \delta \geq t_R(e_2) - t_R(e_1) - s_R(e_1, e_2)$$

This holds because of the assumption that $\delta \leq s_R(e_1, e_2)$. In the second case where $\delta > s_R(e_1, e_2)$, we have

$$\begin{aligned}
\underbrace{t_R(e_2) - s_R(e_1, e_2) + \delta}_{\mathsf{LB}} - t_R(e_1) - \delta \geq t_R(e_2) - t_R(e_1) - s_R(e_1, e_2) \\
t_R(e_2) - t_R(e_1) - s_R(e_1, e_2) \geq t_R(e_2) - t_R(e_1) - s_R(e_1, e_2).
\end{aligned}$$

This shows that in both cases, Equation (4.2.1) holds. For $\delta < 0$, the steps are analogous. For a full proof, it would need to be shown similarly that Equation (4.2.2) holds.

4.2.2 Detection Performance Metrics

Detecting significant timing differences in a set of pairs of time sample distributions can be understood as a classification task: Each pair is classified as either having a significant timing difference (positive example) or not (negative example). We use two classical metrics for evaluating classifiers: Precision and Recall. Precision is the ratio of correctly predicted positives to predicted positives; Recall is the ratio of correctly predicted positives to real positives [31]. Formally, given a set of predicted values P and a set of true values T (cf. [31]):

$$\text{Precision} = \frac{|T \cap P|}{|P|}, \quad \text{Recall} = \frac{|T \cap P|}{|T|}$$

In our case, P is the set of time sample distributions which are marked as significant, while T is the set of distributions which have a fault injected. Since a fault may propagate and influence more time samples than just one, we also keep track of a third set C of *consequential* faults. These are all faults which are not originally injected faults, but inserted by Algorithm 2 in order to keep the schedule consistent. In the example in Figure 4.4, T would include the originally faulty event e_2, and C would include the consequential faults e_3, e_4, e_5. To prevent detected consequential faults from influencing the precision, we modify the above equation:

$$\text{Precision}' = \frac{|T \cap P|}{|P \setminus C|}$$

For our application, a high recall (detecting many of the present faults) is of higher importance than a high precision (low number of false positives): We would rather the engineer has to look at to many potentially significant changes than too few.

Often, the geometric mean of precision and recall, the *F1-score*, is given as a single measure for the performance of a classifier [31]:

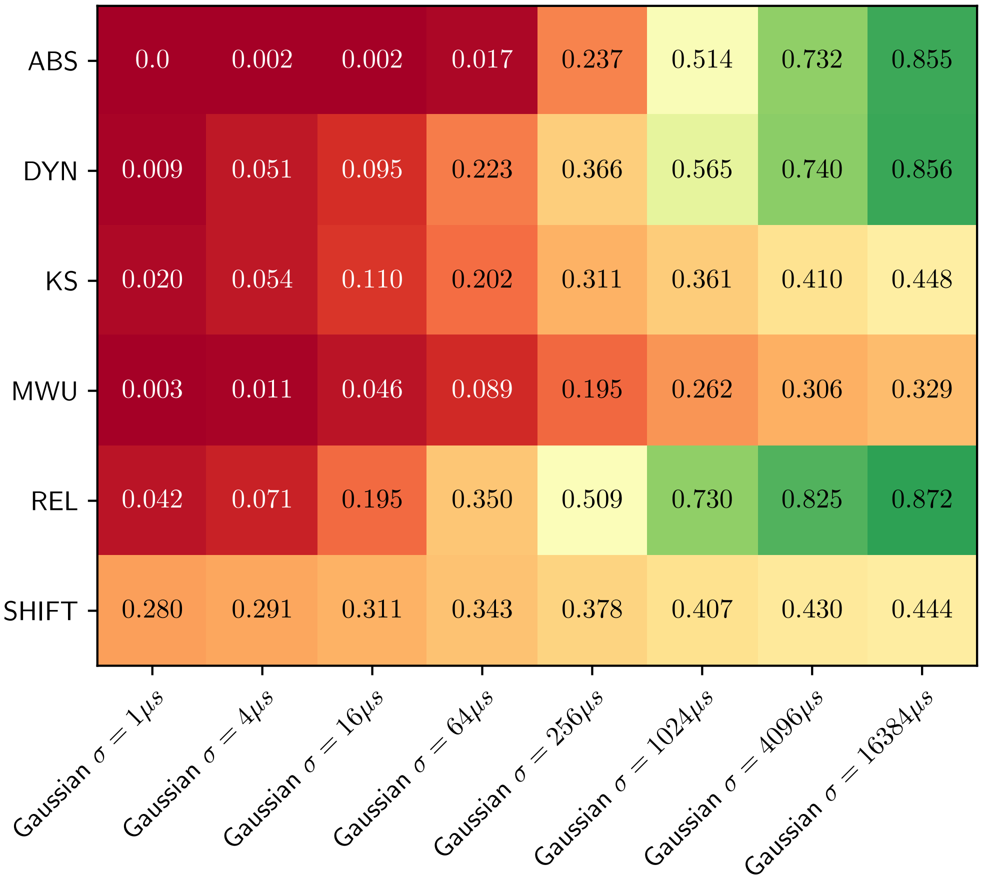
$$\text{F1} = \frac{\text{Recall} + \text{Precision}}{2 \cdot \text{Recall} \cdot \text{Precision}}$$

Note that with these definitions, we require an injected fault to be detected on the *exact* automaton transition that corresponds to the delayed event. This may be too strict, because the true impact of the fault may be localized in a different place – see also the examples given in Section 3.2. Further, note that in experiments where multiple random faults are injected, due to inserting faults one after the other, some faults may cancel out the effects of other faults. We reduce the possible impact of these limitations by running each experiment multiple times with different seeds for the randomly injected faults.

4.2.3 Results

We use a large trace from a component in ASML's "TWINSCAN" line of machines as a reference trace. It contains roughly 1.2 million events. We inject different kinds of faults into this reference trace, which we will detail below. From here on, we use "fault" to refer not only to a single injected fault (e, δ), as above, but also to a set of multiple injected faults which make up one logical fault. Further, we call faults where all such injected delays have the same sign *unidirectional*.

Gaussian Noise. In the first experiment, we add random gaussian noise to the trace by shifting every event in the trace by a small positive or negative amount chosen from a gaussian distribution with mean 0 and standard deviation σ. In the real world, all kinds of intrinsic and extrinsic factors contribute to small variations in each measurement. Generally, we expect our method to be robust against noise. Large deviations could however indicate a timing problem and should be reported. Figure 4.5 shows the results for this first experiment. Each cell shows the recall score for one detection method (y-axis) at one specific fault magnitude (x-axis), averaged across twelve runs. The matrix shows that the hypothesis-test-based methods KS and MWU are most robust against noise, reporting the least detected significant changes: At $\sigma \approx 16\,\text{ms}$, only about 45% (33%) of the



	Gaussian $\sigma = 1\mu s$	Gaussian $\sigma = 4\mu s$	Gaussian $\sigma = 16\mu s$	Gaussian $\sigma = 64\mu s$	Gaussian $\sigma = 256\mu s$	Gaussian $\sigma = 1024\mu s$	Gaussian $\sigma = 4096\mu s$	Gaussian $\sigma = 16384\mu s$
ABS	0.0	0.002	0.002	0.017	0.237	0.514	0.732	0.855
DYN	0.009	0.051	0.095	0.223	0.366	0.565	0.740	0.856
KS	0.020	0.054	0.110	0.202	0.311	0.361	0.410	0.448
MWU	0.003	0.011	0.046	0.089	0.195	0.262	0.306	0.329
REL	0.042	0.071	0.195	0.350	0.509	0.730	0.825	0.872
SHIFT	0.280	0.291	0.311	0.343	0.378	0.407	0.430	0.444

Figure 4.5: Injecting Gaussian noise with varying standard deviation σ. Each cell contains the average recall score of one detection method in 12 experiments.

injected faults are detected as significant by KS (MWU). This is not necessarily advantageous, as the noise at the larger magnitudes ($\approx 1, 4, 16\,\text{ms}$) could already be considered significant timing changes. The Shift method shows a susceptibility to noise, but at a more constant rate, which is independent of the fault size: At $\sigma = 1\,\mu\text{s}$, it already reports 28% of the injected faults, but at the largest magnitude $\sigma \approx 16\,\text{ms}$, it performs comparable to KS at about 45%. This is likely because Shift only considers the location of quantiles, not the magnitude of their deviation. Our combined detection method Dyn's recall is always approximately at the maximum of KS and Dyn, which is due to the "or"-combination of the two detection mechanisms.

Single Spikes. Next, we simulate transient, one-off delays by (a) injecting faults into randomly selected single events ("spikes") and (b) adding delays to a sequence of events in the trace ("chain"). In real-world traces, such faults could for example be caused by temporary resource congestion, network packet loss, or a process being suspended by the operating system. As in the previous experiment, a certain robustness against such single faulty events is desired, but they should be detected if they become very large. Figure 4.6 shows the F1-score for each detection method and each injected fault type. It is apparent that the hypothesis-test-based detection methods mostly do not detect these kinds of faults, where only a single data point in each time sample is changed. Rel and Abs, which were designed for this task, do detect these changes at higher magnitudes, according to how their thresholds are set. Shift performs well on the chain-type faults, but has low precision on the spike-type faults, resulting in lower F1-scores. One could argue that Shift is too sensitive and reports too many low-magnitude faults.

Changed Function Timing. Finally, we consider faults which intend to model software problems: We choose a random function from the trace and modify all its *exit events* in the trace, simulating a change of the function's runtime. The resulting F1-scores are shown in Figure 4.7. In the "delay" case, we simply increase the function runtime observed in the reference trace by a certain factor (For example, for "Delay 1.1x", we delay the exit events by 10% of the time elapsed since the last start event). In the "alternate" case, we perform the same operation, but on every second function execution, we instead *decrease* the runtime by the same factor, to simulate an even more complex timing behavior change. Because all executions of one function are affected in this experiment, the distribution of the time samples are likely to change significantly, which is why the hypothesis-test-based methods KS and MWU perform well here. If the faults exceed a certain absolute magnitude, they are also detected by Abs.

4.2.4 Threats to Validity

The results should be understood as an internal comparison of the different detection methods. While we try to simulate scenarios close to ones observed in real traces, the injected faults are artificial and may be unrelated to faults and regressions occurring in the real world; in particular because the fault injection is done on trace-level and not on software-level.

We intentionally exclude functional differences from this evaluation to prevent modifications to the input trace that would alter its semantics. This restricts the kind of timing behavior changes we can evaluate to purely timing-related changes. One consequence is that the sample sizes of the compared distributions are always unchanged between reference and changed trace.

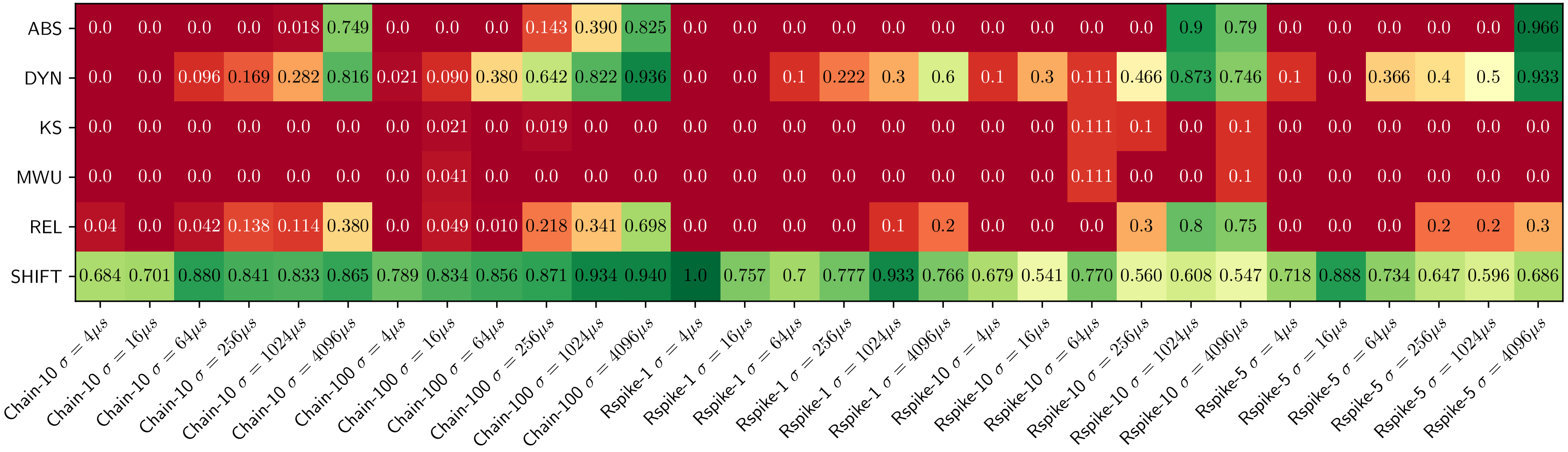


Figure 4.6: F1-scores for the different detection methods from injecting different single spike type faults. Large numbers (green) indicate that a detection methods performs well on detecting a fault type.

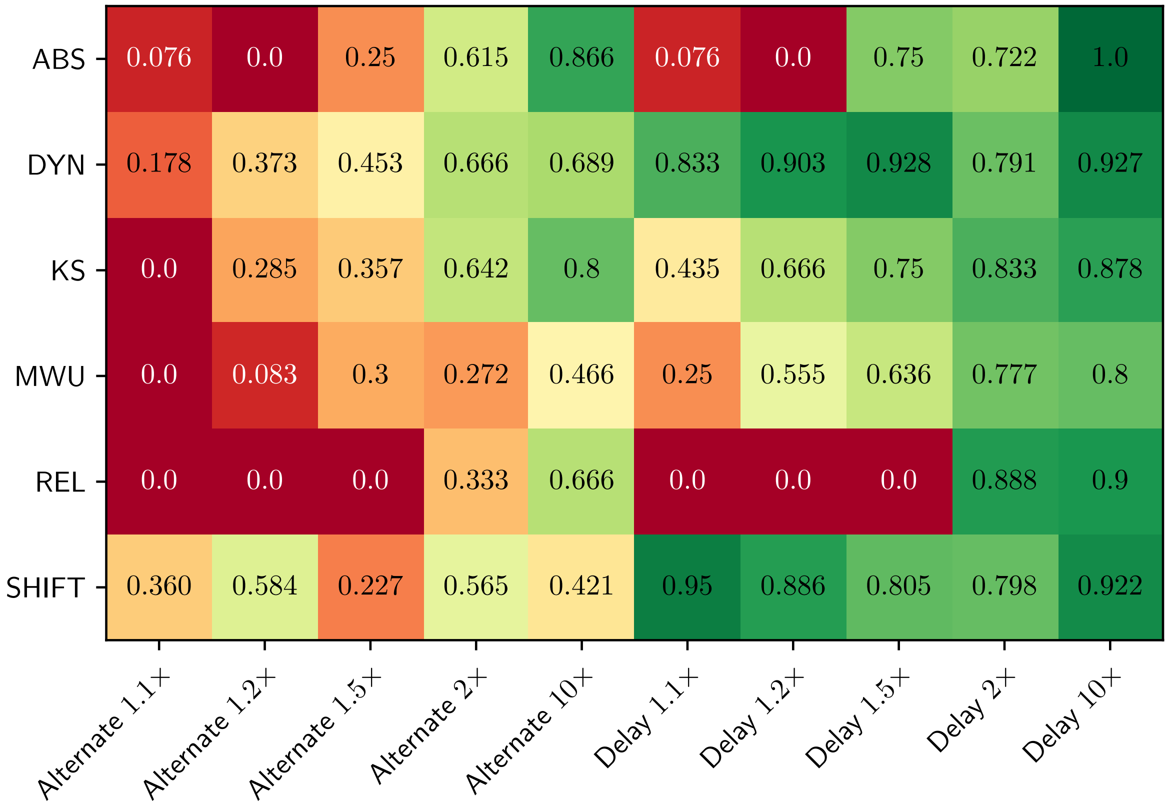


Figure 4.7: F1-scores for the different detection methods from changing function timings.

4.2.5 Summary

The hypothesis-test-based detection methods are less sensitive to noise and outliers, but for the same reasons also miss potentially significant outliers. Abs and Rel catch one-off faults, but can miss low-magnitude faults which only change the shapes of the time sample distributions. The Shift method also seems too sensitive, often reporting many faults but achieving low precision. Relying on Shift could lead to a low signal-to-noise ratio, providing not much value to the engineers. The results are mostly in line with our expectations with the detection methods fulfilling the tasks that they were designed for best and are in that sense complimentary. They also show that the Shift method is not as effective as the others and could potentially be left out.

The experiments show that our detection methods are complimentary. The Shift approach was shown to be not as precise and thus not as effective as the other detection methods. The experiments highlight that designing the detection methods is a balancing act between detecting many faults and not reporting too much noise as significant differences.

4.3 Case Studies with Real-World ASML Traces

In Section 3.6.4, we have already shown results of an analysis with Mids+Time for the non-component-based, open-source software cURL. For more case studies with component-based ASML software, engineers kindly provided us multiple traces from machines or machine test benches.[1]

In three distinct cases, we presented the engineers with our method's results and asked them for feedback. In the first case, the expected behavioral changes were known to us before analysing the traces. For the other two, we did not know about the expected changes beforehand and had the engineers confirm whether the expected changes were found. Notably, we had no knowledge of the structure of the underlying software beyond what we learned from the traces.

4.3.1 Case #1 (Wafer Exposure Controller Regression)

The reference trace for this case is the same that was used for Artificial Fault Injection. The traced component is one of the components responsible for coordinating the wafer exposure in ASML's "TWINSCAN" line of machines. The component suffered from a regression where a component waited for the completion of another task unnecessarily. The bug, which was ultimately found to be caused by a single character code change, caused a delay of more than 200 ms. Mids is already able to detect the regression based on the functional differences, so we only investigate whether including timing data could add more insight.

Our method detects many service fragments with significant changes in their runtime. This includes one change over 370 ms, three changes over 140 ms, and ten changes over 10 ms. Unfortunately, we can not judge in which way these changes relate to the proclaimed 200 ms change. The fact that we cannot directly see this regression may also be due to our method primarily considering the runtime of, and the timings within the service fragments, while the performance regression in this case occurs *between* service fragments.

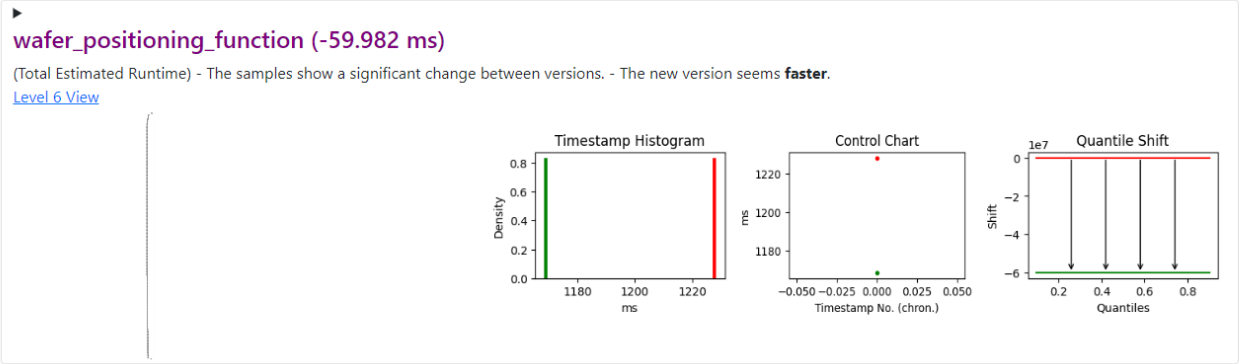
When shown our results, the engineers said they liked the analysis and the visualizations. They noted that while in this particular case, the regression could be found based on just the functional changes, in other cases the added timing information could be useful. They referred us to another group which promised more timing-related issues, which lead to cases #2 and #3.

4.3.2 Case #2 (Wafer Positioning Subsystem Regression)

The traces for our second case study are much smaller than for case #1: The reference trace consists of only 5414 events occurring in less than ten seconds. The most frequently called function is executed 82 times. The trace is a recording of only a single wafer exchange sequence. Most events in the part of the system which is of interest (as determined by our detection methods) are executed only once. We had no prior knowledge of the nature of the timing change that was present in the traces.

Using the overview page of our generated analysis results, we were able to determine that a high-level function ("`wafer_positioning_function`") which is responsible for coordinating tasks in the wafer positioning workflow took ca. 60 ms less time to run to completion in the new trace. Figure 4.8 shows the function as it appears in the overview page – the sorting by significance and purple highlighting made it very easy to spot this timing behavior changes. Its name is colored purple to indicate a significant difference. This function is only called once in the very

[1] All mentioned component- and function names are anonymized in this report.



▸

wafer_positioning_function (-59.982 ms)

(Total Estimated Runtime) - The samples show a significant change between versions. - The new version seems **faster**.

Level 6 View

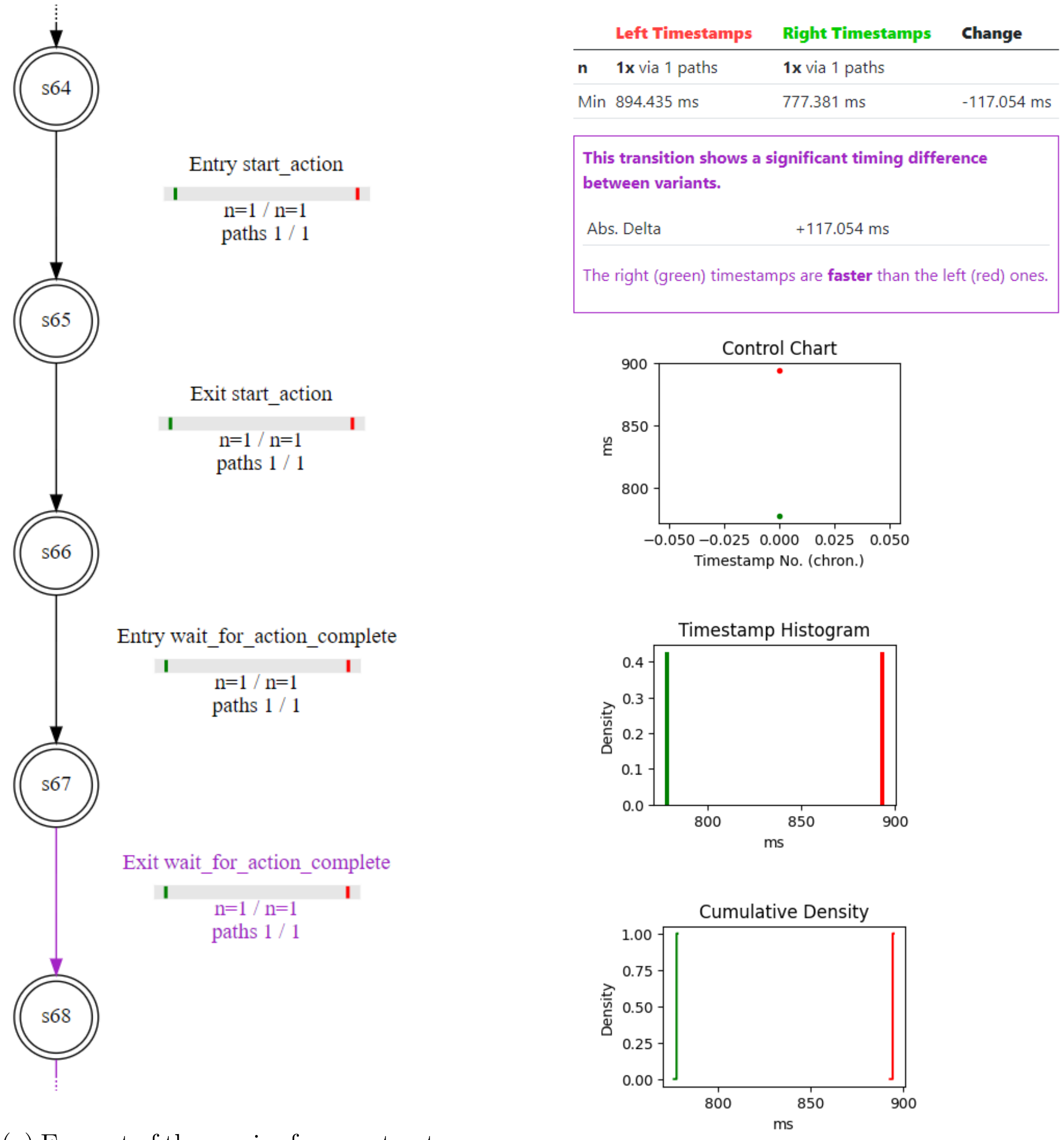
Figure 4.8: Detail from the overview page of our analysis output. The apparent 60 ms decrease in function runtime is highlighted in purple. On the left, a zoomed-out preview of the service fragment automaton is shown. As this is a high-level function responsible for coordinating various tasks, the automaton is large and very linear, making it hard to discern details in this preview. The function is only called once, which results in only two data points being shown in the runtime plots to the right.

short trace, hence the histogram and control chart contain only two data points. By investigating the automaton of this function (Figure 4.9), we can find the exact event which first shows a significant timing difference. Figure 4.9a shows an excerpt of the full automaton, where the exit event of the function "`wait_for_action_complete`" is the first purple transition, indicating a timing difference. Clicking on the transition reveals the information presented in Figure 4.9b, where we can see that a 117 ms difference was detected. In a next step, the service fragment automaton for `wait_for_action_complete` is examined to find at what point the significant difference occured. We can repeat this process to dig deeper into the function call stacks and find the probable cause, or at least determine the event at which the change originates. We were able to determine that two distinct changes caused the 60 ms speed-up in `wafer_positioning_function`: The −117 ms change in `wait_for_action_complete` which was already discussed, caused by an increased time spent waiting on a physical component, together with a +58 ms change in another, similar function. This demonstrates that MIDS+TIME can even be used to analyse timing changes which are not related to software changes.

The engineers confirmed that the +58 ms change was the expected cause of the performance regression. The −120 ms was unexpected and surprised the engineers. They could not say what the cause for the change was, but initially decided to not investigate further after determining that the change seemingly only occurred on a test setup and not in a production machine. They liked the presentation of the results and indicated interest in doing further analyses with the tool, which lead to the upcoming third case study.

4.3.3 Case #3 (Wafer Positioning Subsystem Refactoring)

The third case started out with a set of three trace pairs (six traces in total). They again record only single wafer exchange sequences and are of very similar structure as in case #2. They contained timing differences between −26 ms and +104 ms in the same high-level function mentioned above.



	Left Timestamps	Right Timestamps	Change
n	**1x** via 1 paths	**1x** via 1 paths	
Min	894.435 ms	777.381 ms	-117.054 ms

This transition shows a significant timing difference between variants.

Abs. Delta	+117.054 ms

The right (green) timestamps are **faster** than the left (red) ones.

(a) Excerpt of the service fragment automaton for the function shown in Figure 4.8. It reveals the function call which first caused a timing difference (purple transition).

(b) Additional information which is shown when clicking on the purple edge in (a).

Figure 4.9

Notably, the traces again contained unexpected timing differences of up to +140 ms caused by the same functionality that was already responsible for the unexpected differences in trace #2. This surprised the engineers and led to the decision that this effect should be investigated after all. In this process, the engineers followed up with two more trace pairs that were analysed.

Unfortunately, the investigation of the unexpected timing differences has not concluded at the time of writing.

4.3.4 Summary

The interviewed engineers confirmed our findings and made positive comments about the tool and visualizations, indicating interest in using it for future analyses and continuing research on this matter. While cases #1 and #2 were initiated by us, the engineers independently approached us for the analysis of the traces for case #3 and the traces that were analysed as follow-up.

We saw a wide range of heterogeneous timing data in the real-world traces, which once again highlighted the importance of our different detection methods: While many parts of the investigated systems had low sample sizes, which Abs and Rel can handle, other time samples showed skewed and multimodal distributions which are best handled by KS or MWU.

The detection methods are not a fully automatic system; the reported significant differences not infrequently contain false positives or changes which are not interesting in the context of the investigation. They do however help the engineer to focus on relevant parts of the system.

The case studies show that even people unfamiliar with a software system and its source code can use Mids+Time to analyse its execution traces, gain basic understanding of the underlying business logic, and find and analyse performance regressions. They also demonstrate that it is possible to find and analyse timing behavior changes which are not due to software changes, but for example caused by physical components.

5 Related Work

Our work touches on a variety of topics, including the analysis and visualization of software performance metrics, automata learning, and modelling timing in software systems. In this chapter, we briefly describe a number of works that relate to these topics or approach similar problems as we do.

Timing Behavior Change Correlation on Calling Dependency Graphs. Marwede et al. investigate automatic failure diagnosis in large scale systems [25]. They compute anomaly scores of nodes in the calling dependency graph of a system and use these to determine the probable root cause of a timing behavior anomaly. They focus on the root-cause analysis and not on the computation of the original anomaly scores, but mention that these scores are simply based on the quartiles of function call times. Future work could potentially look into performing root cause analyses on our models with similar strategies as applied by Marwede et al. The authors also discuss the visualization of calling dependency graphs with anomaly scores. Similar to our method, they visualize their results on different abstraction levels, which they call *architecture* levels, and they also add small inline histograms to the graph visualizations.

Anomaly Detection on Control Flow Graphs. Jia et al. present an approach for constructing time-weighted control flow graphs (TCFG) from traces and using them for online (i.e., at-runtime) anomaly detection [20]. Unlike us, they do not assume traces with known well-formed structures, but instead use a probabilistic template-mining approach to extract information from the input trace and construct the TCFGs. For the time-weights stored on the TCFG edges, Jia et al. use the maximum observed execution time between the two incident nodes. Their method can report on different kinds of anomalies, for example when an event from the online log stream exceeds the time weight on the reference TCFG.

Failure Diagnosis on State-Machine Models. Tan et al. derive state-machine models from system traces [37]. While not the main focus of their work, they also look at diagnosing failures by analysing how much time a (possibly defect) system spent at which node compared to other (healthy) systems. They estimate probability density functions (PDFs) from the collected time data using kernel density estimation and then compute the Jensen-Shannon divergence between each pair of PDFs. Pairs which exceed an empirical threshold are diagnosed as potential failures. Similar to our hypothesis-test-based detection methods (KS and MWU), this approach can handle all kinds of different distribution shapes. However, unlike KS and MWU, it is parametric and thus requires finding and setting a good threshold value to be effective.

Timing Behavior Change Detection using Regression Models. Odyurt et al. introduce the concept of a software passport, a kind of timing behavior signature of an execution, which is used for timing behavior change detection in software with repetitive tasks [29]. A software passport is based on a set of regression models for different metrics, such as cumulative CPU time or cumulative I/O event counts. An anomaly is detected when a significant deviation from the regression model is present. Notably, the work of Odyurt et al. has also been conducted in collaboration with ASML.

Time-Series Anomaly Detection. Ren et al. present a time-series anomaly detection approach developed and used at Microsoft, which combines the Fourier-Transform based "Spectral Residual" algorithm and convolutional neural networks [33]. This is quite different from our work, as we did not attempt to find anomalies *within* one time series, but rather compared time data between two different executions. However, it might be interesting for future work to look into this method of detecting machine failures in production, for which the time data we extract from execution traces could be used as a basis.

Timed Automata. In our timing-extended system models, timing data is stored as supplementary data on the transitions of TDFAs. The timing data does not change the semantics or the accepted language of the underlying DFA. There exist abstractions in literature that can model timing behavior in automata, for example Timed Automata, where transitions may only be taken when a set of timing constraints are met [4,6]. For our purpose of comparing and visualizing timings between software versions, these additional modelling capabilities would not have provided much benefit, as our analysis is based more on the structure of the automata rather than their accepted language. This could however be explored in future work.

6 Conclusion

We contribute a methodology for analysing the timing behavior of component-based software systems. It is based on the system models of MIDS, a tool for inferring models from on execution traces and comparing system models of different software versions. We extract timing data from the system's execution traces, and add it to the automata-based system models by annotating the transitions with sequences of timestamps. Based on this data, various metrics, such as a function's runtime, can be computed. We also develop a detection method for timing behavior changes, which combines different simple heuristics to handle noise, non-normality and low sample sizes. Furthermore, we show how the timing data can be visualized intuitively, using different kinds of plots and by extending MIDS' difference automaton concept and its user interface. We evaluate our method on real-world traces of ASML lithography machines, finding it to be capable of helping engineers in detecting real timing behavior differences in a simple and effective manner. Even though we were not familiar with the software systems and the source code that produced these traces, we were able to gain a basic understanding of the system and to find and analyse performance regressions which were previously unknown to the engineers. We received very positive feedback from the engineers at ASML, who approached us independently for additional analyses and showed great interest in integrating our prototype into the main-line version of MIDS. With our proof-of-concept application of MIDS+TIME to the open-source software cURL, we show that our methodology may even be applicable to non-component-based software. Both our experiments with ASML's traces and the cURL trace demonstrate that MIDS+TIME's detection capabilities are not limited to timing differences caused by software changes, but extend to external changes, for example in physical components, as long as they affect the software's execution.

6.1 Future Work

Our method of adding timing data to the models is fairly basic and does not alter the semantics of the underlying automata. Future work could investigate whether modelling the system as Timed Automata, Probabilistic Automata or some other construct would yield any advantages for the analysis. It might also be interesting to investigate whether a kind of root-cause analysis as is done by Marwede et al. [25] can be conducted on the learned models. Our implementation of MIDS+TIME is only a prototype and has room for improvement. Future work could include investigating more real-world cases and collaboration with engineers to tune the detection heuristics to fit the needs of the engineers even better. MIDS+TIME could possibly also benefit from a more interactive user interface, where engineers can define new metrics, set detection thresholds or adjust plots on the fly.

A Appendix

A.1 Instrumentation of cURL

For the screenshots in Chapter 3, we instrument the open-source software cURL to produce traces that can be ingested by MIDS. To this end, we use the GNU compiler's[1] flag for instrumenting functions, `-finstrument-functions`. We log each function entry and exit along with the current time, the function's address, and the caller's address. In a post-processing step, we use `objdump` to reconstruct each function's name from its address and `ctags` to find the compilation unit in which the function was originally defined. To find the calling function's name, we search backwards from the call site to find the start of the enclosing function. We then again use `objdump` to find the caller's name.

This is intended as a proof-of-concept. Finding the compilation unit just based on a function's name is possibly ambiguous, and the instrumentation impacts the timing of the instrumented functions due to the logging overhead.

[1] https://gcc.gnu.org/

Bibliography

[1] A. Akinshin. *Pro .NET Benchmarking*. Springer, 2019.

[2] A. Akinshin. Nonparametric Cohen's d-consistent effect size. https://aakinshin.net/posts/nonparametric-effect-size/, 2020. Accessed: 2022-11-23.

[3] A. Akinshin. Statistical approaches for performance analysis. https://aakinshin.net/posts/statistics-for-performance/, 2020. Accessed: 2022-11-23.

[4] R. Alur, L. Fix, and T. A. Henzinger. A determinizable class of timed automata. In *International Conference on Computer Aided Verification*, pages 1–13. Springer, 1994.

[5] D. Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

[6] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In *Advanced Course on Petri Nets*, pages 87–124. Springer, 2003.

[7] V. W. Berger and Y. Zhou. Kolmogorov-Smirnov test: Overview. *Wiley statsref: Statistics reference online*, 2014.

[8] J. Chen and W. Shang. An exploratory study of performance regression introducing code changes. In *2017 ieee international conference on software maintenance and evolution (icsme)*, pages 341–352. IEEE, 2017.

[9] M. Coffin and M. J. Saltzman. Statistical analysis of computational tests of algorithms and heuristics. *INFORMS Journal on Computing*, 12(1):24–44, 2000.

[10] I. Crnkovic and M. Larsson. Challenges of component-based development. *Journal of Systems and Software*, 61(3):201–212, 2002.

[11] Y. Dodge. *The Concise Encyclopedia of Statistics*, pages 283–287. Springer New York, New York, NY, 2008.

[12] K. Doksum. Empirical Probability Plots and Statistical Inference for Nonlinear Models in the Two-Sample Case. *The Annals of Statistics*, 2(2):267 – 277, 1974.

[13] R. C. Geary. Testing for normality. *Biometrika*, 34(3/4):209–242, 1947.

[14] B. Gregg. Frequency Trails. https://www.brendangregg.com/frequencytrails.html, 2014. Accessed: 2023-01-04.

[15] F. E. Harrell and C. E. Davis. A new distribution-free quantile estimator. *Biometrika*, 69:635–640, 1982.

[16] D. Hendriks and K. Aslam. A systematic approach for interfacing component-based software with an active automata learning tool. In *Leveraging Applications of Formal Methods, Verification and Validation. Software Engineering: 11th International Symposium, ISoLA 2022, Rhodes, Greece, October 22–30, 2022, Proceedings, Part II*, pages 216–236. Springer, 2022.

[17] D. Hendriks, A. v. d. Meer, and W. Oortwijn. A multi-level methodology for behavioral comparison of software-intensive systems. In *Formal Methods for Industrial Critical Systems: 27th International Conference, FMICS 2022, Warsaw, Poland, September 14–15, 2022, Proceedings*, pages 226–243. Springer, 2022.

[18] B. Hooimeijer, M. Geilen, J. F. Groote, D. Hendriks, and R. Schiffelers. Constructive Model Inference: model learning for component-based software architectures. In *Proceedings of the 17th International Conference on Software Technologies-ICSOFT*, pages 146–158, 2022.

[19] M. Isberner, F. Howar, and B. Steffen. The TTT algorithm: a redundancy-free approach to active automata learning. In *International Conference on Runtime Verification*, pages 307–322. Springer, 2014.

[20] T. Jia, L. Yang, P. Chen, Y. Li, F. Meng, and J. Xu. Logsed: Anomaly diagnosis through mining time-weighted control flow graph in logs. In *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, pages 447–455. IEEE, 2017.

[21] H. Jifeng, X. Li, and Z. Liu. Component-based software engineering. In *International Colloquium on Theoretical Aspects of Computing*, pages 70–95. Springer, 2005.

[22] R. Jonk, J. Voeten, M. Geilen, R. Theunissen, Y. Blankenstein, T. Basten, and R. Schiffelers. Inferring timed message sequence charts from execution traces of large-scale component-based software systems. 2019.

[23] T. Lumley, P. Diehr, S. Emerson, and L. Chen. The importance of the normality assumption in large public health data sets. *Annual Review of Public Health*, 23(1):151–169, 2002. PMID: 11910059.

[24] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947.

[25] N. Marwede, M. Rohr, A. van Hoorn, and W. Hasselbring. Automatic failure diagnosis support in distributed large-scale software systems based on timing behavior anomaly correlation. In *2009 13th European Conference on Software Maintenance and Reengineering*, pages 47–58. IEEE, 2009.

[26] F. J. Massey Jr. The Kolmogorov-Smirnov test for goodness of fit. *Journal of the American statistical Association*, 46(253):68–78, 1951.

[27] M. D. McIlroy. Mass-produced software components. In J. M. Buxton, P. Naur, and B. Randell, editors, *Software Engineering Concepts and Techniques (1968 NATO Conference of Software Engineering)*, pages 88–98. NATO Science Committee, Oct. 1968.

[28] T. H. Nguyen, B. Adams, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora. Automated detection of performance regressions using statistical process control techniques. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, pages 299–310, 2012.

[29] U. Odyurt, H. Meyer, A. Pimentel, E. Paradas, and I. Alonso. Software passports for automated performance anomaly detection of cyber-physical systems. In D. Pnevmatikatos, M. Pelcat, and M. Jung, editors, *Embedded Computer Systems*, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), pages 255–268, Germany, Jan. 2019. Springer Verlag. 19th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, SAMOS 2019.

[30] J. Oncina and P. Garcia. Inferring regular languages in polynomial updated time. In *Pattern recognition and image analysis: selected papers from the IVth Spanish Symposium*, pages 49–61. World Scientific, 1992.

[31] D. M. Powers. Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation. *arXiv preprint arXiv:2010.16061*, 2020.

[32] H. Raffelt, B. Steffen, and T. Berg. Learnlib: A library for automata learning and experimentation. In *Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, pages 62–71, 2005.

[33] H. Ren, B. Xu, Y. Wang, C. Yi, C. Huang, X. Kou, T. Xing, M. Yang, J. Tong, and Q. Zhang. Time-series anomaly detection service at microsoft. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 3009–3017, 2019.

[34] G. A. Rousselet, C. R. Pernet, and R. R. Wilcox. Beyond differences in means: robust graphical methods to compare two groups in neuroscience. *European Journal of Neuroscience*, 46(2):1738–1748, 2017.

[35] U. Schöning. *Theoretische Informatik-kurz gefasst*. Springer, 1992.

[36] W. A. Shewhart. *Economic control of quality of manufactured product*. Macmillan And Co Ltd, London, 1931.

[37] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan. SALSA: Analyzing logs as state machines. *WASL*, 8:6–6, 2008.

[38] J. W. Tukey. Comparing individual means in the analysis of variance. *Biometrics*, pages 99–114, 1949.

[39] T. Vale, I. Crnkovic, E. S. de Almeida, P. A. da Mota Silveira Neto, Y. C. Cavalcanti, and S. R. de Lemos Meira. Twenty-eight years of component-based software engineering. *Journal of Systems and Software*, 111:128–148, 2016.

[40] N. Walkinshaw and K. Bogdanov. Automated comparison of state-based software models in terms of their language and structure. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(2):1–37, 2013.

[41] N. Yang, K. Aslam, R. Schiffelers, L. Lensink, D. Hendriks, L. Cleophas, and A. Serebrenik. Improving model inference in industry by combining active and passive learning. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 253–263. IEEE, 2019.

Selbsständigkeitserklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

______________________ ______________________________

Datum **Unterschrift**