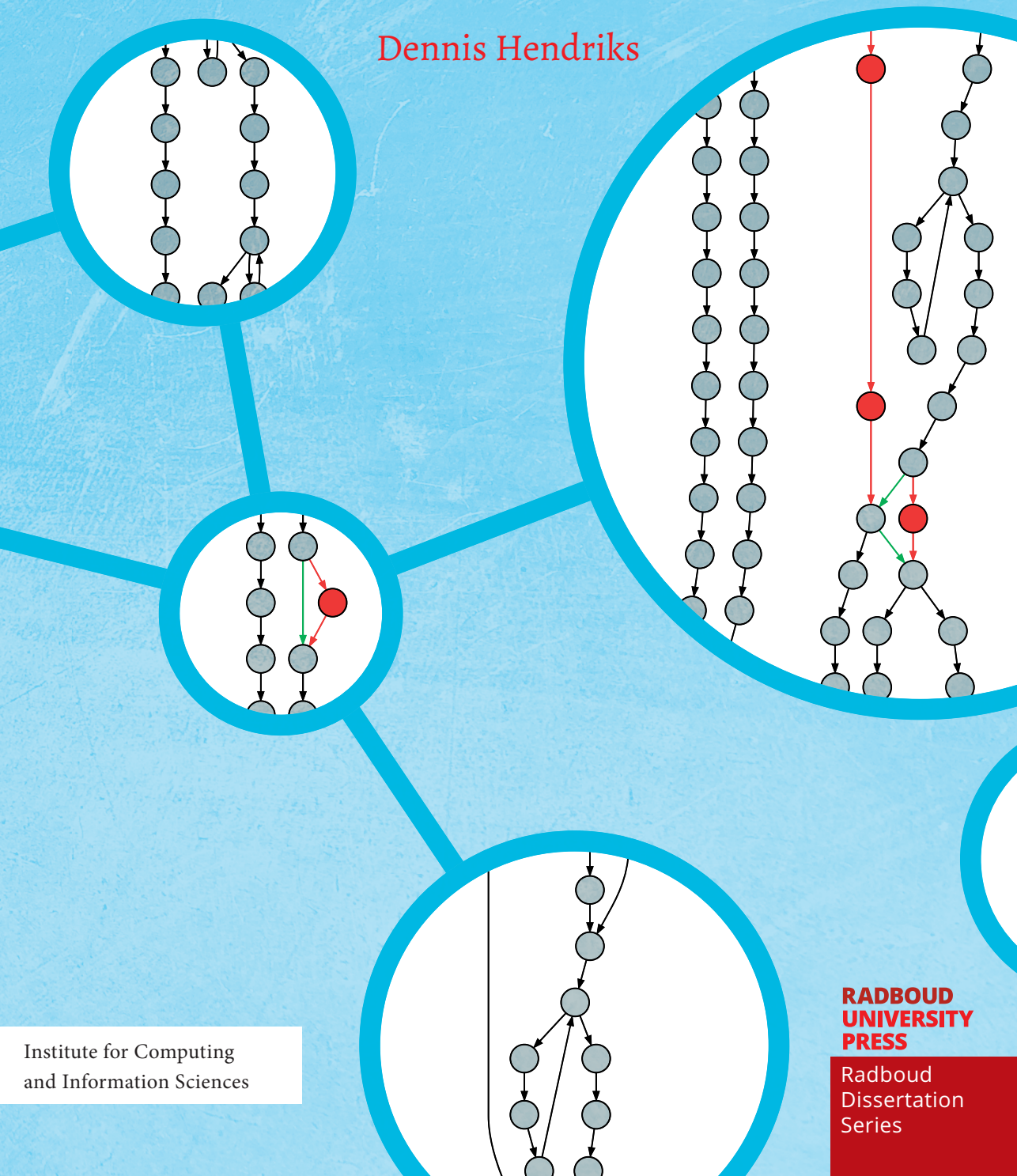
Model Inference and Comparison for Software Evolution in Large Component-Based Systems

Dennis Hendriks



Institute for Computing and Information Sciences

RADBOUD UNIVERSITY PRESS

Radboud Dissertation Series

Model Inference and Comparison for Software Evolution in Large Component-Based Systems

Dennis Hendriks


Radboud University 

The research described in this thesis is largely carried out as part of the Transposition project under the responsibility of TNO-ESI in cooperation with ASML. The research activities are supported by the Netherlands Ministry of Economic Affairs and Climate Policy, TKI-HTSM, and the Radboud University.

Author: Dennis Hendriks
Title: Model Inference and Comparison for Software Evolution in Large Component-Based Systems

Radboud Dissertations Series
ISSN: 2950-2772 (Online); 2950-2780 (Print)

Published by RADBOUD UNIVERSITY PRESS
Postbus 9100, 6500 HA Nijmegen, The Netherlands
www.radbouduniversitypress.nl

Design: Dennis Hendriks and Proefschrift AIO
Cover: Dennis Hendriks
Printing: DPN Rikken/Pumbo

ISBN: 9789493296640
DOI: 10.54195/9789493296640
Free download at: www.boekenbestellen.nl/radboud-university-press/dissertations

RADBOUD
UNIVERSITY
PRESS

Model Inference and Comparison for Software Evolution in Large Component-Based Systems

Proefschrift ter verkrijging van de graad van doctor
aan de Radboud Universiteit Nijmegen
op gezag van de rector magnificus prof. dr. J.M. Sanders,
volgens besluit van het college voor promoties
in het openbaar te verdedigen op

woensdag 28 augustus 2024
om 16.30 uur precies

door

Dennis Hendriks
geboren 15 juli 1984
te Nijmegen

Promotor:
prof. dr. F.W. Vaandrager

Copromotor:
dr. ir. G.J. Tretmans

Manuscriptcommissie:
prof. dr. J.H. Geuvers
prof. dr. J.J.M. Hooman
prof. dr. M. Huisman (Universiteit Twente)
dr. K. Bogdanov (University of Sheffield, Verenigd Koninkrijk)
prof. dr. ir. J.F. Groote (Technische Universiteit Eindhoven)

Summary

Large, complex systems are often divided into components. Engineering teams work on their components in relative isolation, making them less aware of what goes on in the rest of the system. As the system and its software inevitably evolve, engineers must understand the (communication) behavior of the current software to make proper changes to it, and they must understand the impact of their changes to prevent costly regressions and reduce risks.

To make proper changes, engineers require an overview of the software communication behavior. For large component-based systems, obtaining and understanding this overview is both time-consuming and error-prone, even with the available source code, tests, and documentation. We therefore (semi-)automatically obtain the behavior using model inference techniques, and capture it in suitable state machine models that provide the overview. We first use active automata learning to infer models by querying the system. To reduce the effort required to set up active automata learning, we develop a systematic approach to connect software components to a learning tool, dealing with the various types of (a)synchronous interactions with the isolated component code. As inferring models is then still too time-consuming, we use state machine learning to more quickly infer models, from execution logs. To overcome the limitations of existing heuristic-based state machine learning algorithms, such as hard-to-configure settings and over-generalization of the behavior, we develop the Constructive Model Inference approach. It infers intuitive multi-level models that match the structure of component-based systems.

When changes are made, engineers often do not understand the system-wide impact of those changes. We therefore infer models of the software versions before and after a change, comparing them to find all behavioral differences. Our multi-level comparison approach integrates multiple complementary methods to automatically compare the behavior of state machine models. Comparison results are presented at six levels of abstraction, providing engineers with a good overview, and guiding them through the relevant differences by step-by-step zooming in, without wasting time on unaffected parts of the system.

We evaluate the model inference and comparison methodologies using various case studies at ASML, a company that develops and manufactures lithography systems. Our open-source MIDS tool combines the methodologies. MIDS is extensible, allowing it to be applied at other companies to similarly ease their software evolution challenges.

Samenvatting

Grote, complexe systemen worden vaak verdeeld in componenten. Teams werken relatief geïsoleerd aan hun componenten, waardoor ze minder afweten van wat er zich in de rest van het systeem afspeelt. Naarmate het systeem en de bijbehorende software onvermijdelijk evolueren, moeten ingenieurs het (communicatie)gedrag van de huidige software begrijpen om er correcte wijzigingen in aan te brengen, en moeten ze de impact van hun wijzigingen begrijpen om kostbare regressies te voorkomen en risico's te beperken.

Om de software correct te wijzigingen, hebben ingenieurs een overzicht nodig van het communicatiegedrag van de software. Voor grote component-gebaseerde systemen is het verkrijgen en begrijpen van dit overzicht zowel tijdrovend als foutgevoelig, zelfs met de beschikbare broncode, tests en documentatie. We leren daarom het gedrag (semi-)automatisch met behulp van modelinferentie technieken, en vangen het gedrag in geschikte toestandsmachine modellen die het overzicht bieden. We gebruiken eerst het actief leren van automaten, waarbij modellen worden geleerd door het systeem te bevragen. Om het actief leren van automaten met minder inspanning in te kunnen zetten, ontwikkelen we een systematische aanpak om softwarecomponenten te verbinden met een leertool, dat kan omgaan met de verschillende soorten (a)synchrone interacties met de geïsoleerde componentcode. Omdat het leren van modellen dan nog steeds te tijdrovend is, gebruiken we het passief leren van toestandsmachines om sneller modellen te leren, uit executielogs. Om de beperkingen van bestaande heuristische passief leren algoritmes te overkomen, zoals moeilijk te configureren instellingen en overgeneralisatie van het gedrag, ontwikkelen we de 'Constructive Model Inference' aanpak. Het leert intuïtieve meerlaagse modellen die overeenkomen met de structuur van componentgebaseerde systemen.

Wanneer er wijzigingen worden aangebracht, begrijpen ingenieurs vaak niet de systeembrede impact van die wijzigingen. We leren daarom modellen van de softwareversies voor en na een wijziging, en vergelijken deze om alle gedragsverschillen te vinden. Onze meerlaagse vergelijkingsaanpak integreert meerdere complementaire methoden om het gedrag van toestandsmachine modellen automatisch te vergelijken. Resultaten worden gepresenteerd op zes abstractieniveaus, die ingenieurs een goed overzicht geven, en ze door de relevante verschillen leiden door stapsgewijs in te zoomen, zonder tijd te verspillen aan onveranderde delen van het systeem.

We evalueren de modelinferentie- en modelvergelijkingsmethodologieën in verschillende casestudies bij ASML, een bedrijf dat lithografiesystemen ontwikkelt en produceert. Onze open-source MIDS-tool combineert de methodologieën. MIDS

is uitbreidbaar, waardoor het bij andere bedrijven kan worden toegepast om op vergelijkbare wijze hun software-evolutie uitdagingen te verlichten.

Acknowledgements

My PhD would not have happened, or would not have had the result it now has, if not for the many people that I worked with over the years, and that have supported me.

First of all, I would like to thank my promotor Frits Vaandrager, and copromotor Jan Tretmans. Thank you for the opportunity to do my PhD in a somewhat unusual way. Thank you for the nice discussions and critical reviews. Thank you for all your support.

Secondly, I would also like to thank Wouter Leibbrandt, my manager at TNO-ESI for many years. Thank you for helping me grow, proposing me for a university position, providing me the opportunity to do a PhD, and the great support you have shown me throughout the years. Thanks also to Ronald Begeer, who took over from Wouter as my direct manager at TNO-ESI, for your interest and support.

I'd also like to thank ASML, for supporting the Transposition project, for our collaboration on it, and for supporting my work in particular. Thanks especially also to Ramon Schiffelers of ASML, for the intense collaboration through the years, among others in the Transposition project, and for helping me to grow, and for supporting me in my work.

Furthermore, I want to thank Wytse Oortwijn of TNO-ESI, for working together on the Transposition project for years, for the great work we did together, for being a trusted colleague, for being a team, and for having fun doing it.

I want to also thank Jeroen Voeten, for being a colleague at TNO-ESI for many years. You may no longer work for TNO, but while you did, you were a great mentor to me. Thanks for our collaborations at TNO and ASML, for your support, and for helping me grow.

Thanks also to the various other colleagues – researchers, scientific programmers, and project managers – that worked with me in the Transposition project, and its predecessor 'Model inference' track in the Concerto project, for the collaborations over the years: Arjan van der Meer, Bas Huijbrechts, Harm Munk, Jacques Verriet, Jeroen Voeten, Jos Hegge, Leonard Lensink, Luna Luo, Mladen Skelin, Paul Rijnvos, Rolf Theunissen and Thomas Nägele.

And thanks to the colleagues in the Concerto and Maestro projects of TNO-ESI and ASML, for our collaborations over the years, in particular: Yuri Blankenstein, Rolf Theunissen, Kostas Triantafyllidis, Sobhan Niknam and Niko Stotz.

Thanks also to the various MSc and PhD students that I worked with over the years on topics related to this thesis: Aaron Hilbig, Berk Aksakal, Bharat Garhewal, Bram Hooimeijer, Jasper Premchand, Kousar Aslam, Leon Freriks, Maikel Leemans, Nan Yang and Ruben Jonk. Thank you for your contributions.

Thanks also to their various university supervisors, for the collaborations.

And thanks to my managers at ASML through the years, Ramon Schiffelers, Hans Onvlee and Jacek Kustra. Thanks for our collaborations, quickly reviewing my papers so that I could submit them to ASML's Technical Publication Board on time, for helping with bureaucratic matters, and for your support in general.

Thanks also to the various technical competence leads at ASML, and other technical stakeholders that were at one point involved with the Transposition project, most notably: Arthur Swaving, Dennis Verhaegh, Ivo ter Horst, Jan Friso Groote, Leon Kleikers, Sven Weber and William van Houtum. Thanks for your collaboration and support over the years.

And thanks to the dozens of engineers, architects and testers at ASML that I collaborated with on dozens of case studies over the years. Thanks for making time, trying out my work, and giving feedback.

Thanks also to Theo Engelen, Henk Kant, Gernot Eggen and Frank Schuurmans of ASML, and Ronald Begeer, Frans Beenker and Jacco Wesselius of TNO-ESI, for your support of the Transposition project and my work.

And thanks to Diego Damasceno for the collaborations on structural comparison, while you worked at the RU. It was great to see you recreate some of your earlier work using gLTSdiff, applying my work in a different context.

Thanks also to all the people that were at one point or another involved in the industrialization of my work at ASML, most notably: Alok Lele, Erwin de Hart, Ignacio Alonso, Li Wei Hu, Ruud Theunissen, Sandu Zazu, Suzan Sahin, Sven Weber, Wilbert Alberts and Yuri Blankenstein.

I would also like to thank the various (anonymous) reviewers of the papers on which my thesis is based, the members of the manuscript committee for my thesis, as well as the people that I met at the conferences that I visited. Thanks for your feedback on my work, which allowed me to improve it.

Furthermore, I would also like to thank the various other colleagues that I worked with over the years at TNO-ESI, RU and ASML. Thanks for the nice collaborations.

And I'd like to thank my family, especially Rieky, Jan, Chantal and Bart, for their interest in my work, and for their support.

Finally, I'd like to thank all the other people that I worked with, or that supported me over the years.

Contents

Chapter 1

Introduction

High-tech embedded and cyber-physical systems are becoming increasingly complex with each generation. Consider for example systems produced by ASML[1], a company that develops and manufactures lithography systems for the production of computer chips. Figure 1.1 shows three generations of their system, from 1984 to 2020 (each at a different scale, but note the number of vertical 'segments'). However, these systems do not only grow in size physically. Newer generations also contain more software.

To handle the increasing complexity of their systems, companies typically use a divide-and-conquer approach. By introducing a component-based software architecture [119], systems are divided into components and sub-components, each with their own responsibilities. The division into smaller and smaller sub-components continues until the components are small enough to be handled by individual teams. Each team can then develop their own component in isolation, at their own pace, without having to know all the details of the rest of the system. For example, ASML's TWINSCAN systems employ such a component-based software architecture, and contain hundreds of software components.

However, introducing a component-based software architecture leads to various challenges of its own [125]. In this thesis, we focus on some of the particular challenges that arise as systems evolve [84, 132, 134]. Over time, systems are

[1] See `https://www.asml.com`.

Figure 1.1: Three generations of ASML's TWINSCAN system. Source: ASML. © ASML.

adapted to cater to changing user requirements, and to improve their performance, reliability, accuracy, and so on. Software engineers add new features and fix bugs, such that at the end of the day the software is different from what it was when the day started. In this context of evolving software in component-based cyber-physical systems, we focus on the following two challenges, and how they can be addressed:

1. **Understanding the current software behavior:** Whenever engineers have to make changes to the software of a system, they must first understand the current system and its software. After all, how can they be expected to make proper changes to the software, if they don't understand it? However, as we will see, understanding how large component-based systems behave is far from trivial. Therefore, engineers spend considerable amounts of time gaining this understanding [30, 37, 78, 132, 134].

2. **Understanding the impact of software changes:** Once engineers understand the software of the current system, they change it. They must then ensure that their changes are correctly implemented, as incorrect changes may break the system. This too is far from trivial, as the impact of the software changes on the system behavior is not always sufficiently clear, especially for large component-based system. Hence, software changes may introduce regressions, making software evolution risky [112, 127, 132].

In this thesis, we opt for a model-based approach. We automatically infer models of software (components). These models provide insight into how the software behaves, such that proper changes can be made. We also automatically compare such software behavior models, for instance of software versions before and after a change. This gives insight into the effects of the changes on the software behavior, to understand their impact and prevent regressions. The model inference and model comparison approaches are combined into a single methodology. Together, they help to make engineers more efficient, and to reduce the risks for software evolution. The approaches are integrated into an open-source software tool [120], to make our methodology widely applicable.

The remainder of this chapter is structured as follows. First, we take a closer look at our study context: systems with a component-based software architecture, and in particular the evolution of the software of such systems (Section 1.1). Then, we also take a closer look at the two challenges (Section 1.2). Subsequently, we discuss our problem statement and research questions (Section 1.3). After that, we present our methodology (Section 1.4). Then we discuss our contributions, and how these address the research questions (Section 1.5). Finally, we outline the structure for the remainder of this thesis, including how the various chapters relate to the contributions, and to previously published papers (Section 1.6).

1.1 Study context

We discuss in more detail our study context and scope. First, we discuss in Section 1.1.1 the type of systems that we consider: cyber-physical systems with a

component-based software architecture. Then, building upon that, we discuss in Section 1.1.2 the evolution of the software of such systems. To some extend we justify our scoping choices in this section. However, we come back to this later as well, as we further discuss the effects of these choices in Chapter 7, where we reflect on our work.

1.1.1 Component-based software architectures

As already mentioned, we consider cyber-physical systems. In particular, we consider high-tech cyber-physical systems that are both large and complex, such as lithography systems from ASML. The software of these systems is often developed using Component-Based Software Engineering (CBSE). The system then has a component-based software architecture [38, 91, 119], where the system is divided into many small *components* that can be developed and maintained in isolation. Especially if the systems are large and complex, CBSE can lead to increased productivity, higher quality, and reduced costs [8, 124].

However, the components of such a system do not exist only in isolation. Most components depend on functionality provided by other components to realize their own functionality. Components therefore *communicate* with each other via *interfaces*, for instance to request certain *functions* to be executed, to exchange information, or to inform each other of their progress in executing their functions at the request of other components.

We focus on component-based systems with a client-server architecture [101], as shown in Figure 1.2. In such systems, for every communication two components are involved, taking the roles of *client* and *server*. Typically, the client initiates the communication, by sending a *request* to the server. The server receives the request, processes it, and responds by sending back a *reply* to the client. Generally, if a component is not processing a request from one of its clients, it is idle. A component can be both a server to its clients (providing functionality) and a client to its servers (requiring functionality), like Component 2 in Figure 1.2.

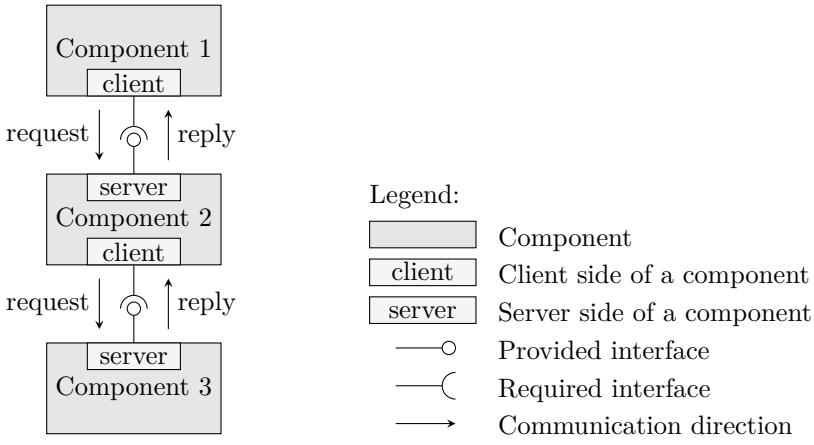


Figure 1.2: Three communicating components, with their interfaces and roles.

Components may communicate both *synchronously* and *asynchronously* [38, 53]. When components communicate synchronously, the client is blocked while awaiting the server's reply. When they communicate asynchronously, the client is free to perform computations and other communications while awaiting the server's reply. Asynchronous communication thus allows components to execute *concurrently*, allowing the system to achieve its performance requirements [92].

The system as a whole then consists of many components that interact, to provide the system with its intended functionality. The communications between two components, as well as the order in which they may occur, may be formalized in interface *protocols* [80, 104]. Together, all the combined interactions between the various components of the system, form the *communication behavior* of the system [96]. This behavior can become quite complex, as many different sequences of requests and replies are possible. This complexity is especially prevalent in component-based systems with asynchronous communication, where the communications of the concurrently-executing components may *interleave*, leading to many similar sequences with subtly reordered communications [90, 104]. There, the number of possible interaction sequences quickly grows to the number of grains of sand in a desert.

1.1.2 Software evolution

Real-world component-based cyber-physical systems (such as those developed by ASML) consist of many components, and therefore a large amount of software. Such large codebases inevitably evolve over time, as the system is continuously adapted to cater to changing user requirements, and to improve its performance, reliability, accuracy, and so on [85].

As a system *evolves*, its specification, design and (software) implementation are changed time and again through successive *changes*. In this thesis, we primarily take the perspective of software engineers, and therefore focus on the software implementation. We consider software implementation changes in the broad sense, from a single line code change to the replacement of an entire legacy component, and from fixing small bugs to implementing entirely new features.

Such changes may affect not only the functionality of the system, but also its other characteristics, such as its performance. We focus primarily on software functionality, and consider the non-functional aspects of software changes to be out of scope. We believe that if we are successful in managing the complexity of functional software changes, we can already have a big impact on reducing the challenges for software evolution in industry (as further outlined in Section 1.2). Furthermore – as the rest of this thesis shows – by considering the software functionality, we may at times also be able to gain insights into, and prevent issues relating to, non-functional aspects.

We further scope our work by focusing on the *impact* of software changes on the communication behavior of the system. Changes have impact if they cause *differences* in which messages are exchanged between components, or the order in which they occur. This choice is based on our experience in working with various companies in the Dutch high-tech industry, and the challenges they face, ASML included (as further discussed in Section 1.2).

We primarily aim to prevent *regressions*, unintended changes in the functional

behavior. For instance, if a new version of a component is meant as a drop-in replacement, we aim to find any changes in the communication behavior of this new component compared to the previous version, in terms of how they communicate with the rest of the system. However, we also consider qualification of *progressions*, intended changes of the functional behavior. For instance, new messages may be exchanged if new functionality is added to the system, and these communications should function as intended. The new functionality should by itself function properly, but could also lead to unintended changes to the existing functionality; regressions and progressions are thus closely related.

For practical reasons, we primarily consider *control components*, (high-level) components that orchestrate various other (lower-level) components [15]. The behavior of such components is predominantly determined by the messages they exchange, rather than for instance the data that they process. In fact, at ASML new control components are developed through model-driven engineering, and certain modeling formalisms do not even allow modeling of data-dependent behavior [28]. Our focus on control components allows us to leave data mostly out of scope. But, even though data is thus mostly out of scope, and we focus on the order of exchanged messages, this does not preclude us from observing the effects of data on the software behavior, as we will see in the remainder of this thesis.

Note that we consider the software of the system, not the entire system. However, the software is an important part of a cyber-physical system (the 'cyber' part). Changes to the functionality of the software are therefore likely to have an effect on the functionality of the entire system as well, especially changes in higher-level components. Hence, we use the software also as a lens that to some extent allows us to consider the system behavior.

1.2 Challenges

We discuss in more detail the two challenges that are the focus of this thesis. We start in Section 1.2.1 with the challenge of understanding the current software behavior, especially in the context of large component-based systems. Building upon that, we discuss in Section 1.2.2 the challenge of understanding the impact of software changes, also in the context of large component-based systems.

1.2.1 Challenge 1: Understanding the current software behavior

In practice, different components developed by independent teams are combined to form sub-systems, which in turn are combined to form the complete system. The *integration* of components to form (sub-)systems often reveals issues, both in the individual components and their composition, among others due to the complexities of inter-component communication [27, 80, 87]; and concurrency and interleaving certainly don't help.

For the integration to be successful despite these complexities, an engineering team must have a good understanding of the interactions between their own component and the other components that their component communicates with [132]. Only by understanding the interactions, can software engineers ensure that the

components they develop can correctly 'talk' to each other, ensuring correct interactions and smooth integration.

For engineers however, it can be quite challenging to get a good overview of how components communicate. If they want to understand how their component communicates with other components, the information they need is typically not found in a single place. Hence, they consult multiple sources of information, such as source code, tests, execution logs, documentation and domain experts, to form their own mental picture. However, even then it is unfortunately still a challenging task, as often: source and test code are written in various programming languages, and contain too many irrelevant details; tests, logs and documentation are incomplete; logs are too long and contain too many slightly different repetitions of the same behavior; documentation becomes outdated; and domain experts are either not available, or they have insufficient time. Yang et al. interviewed 25 engineers and architects at ASML, detailing many of these practices and challenges [132]. However, these challenges are not unique to ASML, as other companies face them as well [30, 111, 127, 131].

Over time and through painstaking effort, engineers do build up a mental picture of the communication behavior [30, 131]. As new people join, they require quite some time to get up to speed and to become productive members of the team. And all the while, components change, as do the components they communicate with. Engineers thus need to keep their mental picture up-to-date, or they risk making decisions based on outdated information.

Getting a good overview of the system behavior is especially difficult for large component-based systems [132]. On the one hand, engineers develop and maintain their own component, which is just one of the components of the larger system. If the entire system is a giant puzzle, they work on only a single piece of that puzzle, but they know what that piece looks like, having worked on their own component for years, fixing bugs, and adding features.

On the other hand, they are often much less aware of the puzzle pieces at the other side of the puzzle [106]. This is good, as the whole point of a component-based software architecture is to have small components for teams to work on in isolation. Interfaces decouple the components. An interface alone should provide all the information that is needed to communicate with another component. The interface then serves as a contract between communicating components [31], a mutual agreement on how they will interact, that abstracts away the internal details of the other component. Engineers then do not have to look through the source code of another component to understand how to communicate with it.

In practice however, an interface contract rarely provides enough information. Typically, the syntax of the interface is described and well-understood, as the functions, their arguments, and types are described in for instance C header files or using an Interface Description Language (IDL). It is thus clear what functions are available for communication. Other aspects of the interface contract, such as the *protocol* that describes the order in which the functions may be called, often get much less attention [27, 80]. The more complete the interface contract is, the more it serves as a proper abstraction, shielding engineers from the internal details of other components.

If the interface contract is not complete enough, the engineers will still go looking for the missing information [127]. They may for instance scour through

the source code of various components that directly communicate with their own component, and even recursively through the code of components connected to those components, and so on. Without proper abstraction, there is no telling how intertwined the components have become, where the relevant information can be found, and how much time it will take to find it.

1.2.2 Challenge 2: Understanding the impact of software changes

Even if software engineers understand the current behavior of their software, changing that software is still risky. Engineers must ensure that their changes are correctly implemented, as otherwise these changes may break the system, which can have serious repercussions.

Consider for instance a team of engineers working on a legacy component that is implemented with end-of-life technology. They are tasked with re-engineering the component to use the company's new technology of choice. Once the new version of the component is ready, it will be integrated into the system, replacing the old version of the component. This new component must then still communicate with its environment in the same way it did before [111], as its clients will still send the same requests and expect the same replies. If the new component has different behavior, and is thus not backwards-compatible [104], this may cause a cascade of regression problems upon integration: the new component may reply to requests differently; its clients may crash if they are not equipped to handle the different replies; the clients will then no longer be able to reply to their clients; and so on. In the end, this may cause the entire system to no longer be able to perform its intended function. The system may then experience downtime, which can be very costly. For instance, at ASML's customers downtime can cost thousands of euros per minute due to productivity loss [14]. Engineers must thus prevent behavioral regressions when changing software, regardless of whether it involves small patches or complex redesigns.

To prevent regressions, it is essential to understand the impact of the changes on the software behavior [86, 127]. If the impact is unclear, it is impossible to anticipate whether the system may break, and to prevent that. It is like taking down a wall of a house to replace it with a new wall: if you do not know whether it is a load-bearing wall, you do not know whether taking it down will bring down the house.

Unfortunately, the impact of software changes is not always clear, especially in large component-based systems. Engineers work on their own components, which are part of the larger system. While engineers may know how their changes impact other close-by components, they may lack the understanding to know how these changes impact further-away components [54, 86, 88, 133]. A small local change may then lead to subtle changes in seemingly-unrelated far-away components, potentially breaking them. Engineers lack an overview of the impact of their software changes on the software's communication behavior.

With sufficient time, such issues may be prevented. For instance, engineers may perform extensive tests [4, 87, 127]. However, in practice often limited tests are available, with insufficient coverage of the software behavior [111]. Hence, even with testing, it may be challenging to determine the impact of software changes

on the software's communication behavior. Then it is still too difficult to gain sufficient confidence in the correctness of the changes, to prevent regressions and reduce the risks.

1.3 Problem statement

Given the challenges described in the previous sections, we can conclude that software evolution is a challenging endeavor. It requires considerable effort and is risky, especially in large component-based systems. Therefore, the main research question for this thesis is:

RQ How can we reduce the efforts and risks of software evolution, for large component-based systems?

We can divide this main research question into two smaller research questions, RQ-1 and RQ-2, that scope our work. This division closely relates to the two challenges described in previous sections, as RQ-1 follows from Section 1.2.1, and RQ-2 follows from Section 1.2.2.

Recall that, in Section 1.2.1, we discussed that to make correct software changes it is essential to understand the current communication behavior of the software, but it is difficult to gain this understanding. Software engineers lack the overview, as there is no single place where they can find all the information. Manually scraping it together is challenging as well, and it is a moving target as the software changes all the time. In large component-based systems, the task is even more daunting. Engineers only work on their own part of the system, and they lack interfaces with clear contracts that include interface protocols. Hence, we formulate RQ-1:

RQ-1 How can we efficiently obtain a complete overview of the software communication behavior, for large component-based systems?

Also recall that, in Section 1.2.2, we discussed that even if software engineers understand the current communication behavior of the software, it is still too difficult to gain sufficient confidence in the correctness of changes, prevent regressions, and reduce the risks. Engineers lack an overview of the impact of their software changes on the software's communication behavior. Again, this is even more challenging in large component-based systems, where engineers work on only a small part of the system. It is unclear to engineers what other behavioral changes are caused by their local changes, and how such changes affect different parts of the system. Hence, we formulate RQ-2:

RQ-2 How can we efficiently obtain a complete overview of the system-wide impact on the communication behavior caused by software changes, for large component-based systems?

Next, we look at both RQ-1 and RQ-2 in more detail, by discussing existing solutions to address these challenges, and identifying gaps. Then, each of these research questions is further divided into two even smaller ones.

1.3.1 RQ-1: Software behavior overview

For research question RQ-1, the aim is to efficiently obtain a complete overview of the software communication behavior, for large component-based systems. In Section 1.2.1 we discussed that it is quite challenging and time consuming for engineers to get such a complete overview. We therefore aim to assist them in obtaining this overview, by automatically extracting the relevant information, and by presenting that information in such a way that it can be easily understood.

Let us first consider the latter, the representation of the information. In order for engineers to be able to understand the software communication behavior, it must be represented in a suitable way. For instance, the set of all possible sequences of interactions is not very suitable. It would likely be very large, and contain many long sequences (they could even be infinite), making it difficult for engineers to get the overview. Therefore, often *models* are used to represent the behavior in a more compact way, and allow engineers to more easily understand it. Examples of such models are state machines and Petri nets.

Let us then consider the former, the automatic extraction of the relevant information, to be captured in those models. We explicitly choose to extract information from existing artifacts, rather than to improve those artifacts themselves. This choice stems from the fact that large component-based systems consist of many components, developed by many teams. It would not be feasible to get all the teams to manually improve their artifacts to the level that would be required for our purposes, especially not in the short term. For instance, manually modeling the behavior of all components of the system could easily take years, or even decades [17]. Furthermore, the individual artifacts of the various components also do not provide a sufficient overview of the interactions between the different components. Hence, our desire is to automatically extract the relevant information, for all the components in the system, and to properly capture the interactions between those components in models.

We therefore turn to software behavior *model inference*, a collection of techniques to automatically infer behavioral models for a software system [123]. These techniques are well-suited for our needs, as they can help engineers in their task to understand the software behavior, by determining the relevant information about the communications of software components. Tools that implement the model inference techniques provide automation, which reduces the time that engineers have to invest to determine all the possible component interactions. Furthermore, new up-to-date models can automatically be extracted again after the software changes, once more saving time.

Next, we look at some related literature on software behavior model inference. Then, we identify existing classes of model inference techniques, and discuss their pros and cons, in relation to our context and the goals of our work. Based on that, we further scope our work by selecting sub-classes of techniques that we investigate in this thesis. Specifically for those techniques, we identify the gaps that we address in this thesis. And we use those gaps to refine research question RQ-1, by further dividing it into two smaller research questions.

Related literature

Software behavior model inference is a well-studied field, although it is not always called as such in the literature. It is for instance studied in the field of software *reverse engineering*. In their seminal work, Chikofsky and Cross divide the reverse engineering task into two parts, as we do, by defining reverse engineering as "the process of analyzing a subject system to identify the system's components and their inter-relationships, and create representations of the system in another form or at a higher level of abstraction" [34]. They also contrast reverse engineering against *reengineering*, by considering reverse engineering to be a process of examination, and reengineering to be a process of change. They do note that reengineering generally includes some form of reverse engineering, prior to making changes.

In a more recent work, Canfora et al. provide an overview of the reverse engineering field [30]. Similar to our work, they identify the need to understand the software before it can be changed, especially as the complexity of the system grows. They define software reverse engineering as "a broad term that encompasses an array of methods and tools to derive information and knowledge from existing software artifacts and leverage it into software engineering processes". We look at these methods and tools below, after discussing further related literature. Confora et al. also mention distributed services as a trend, specifically in the context of service oriented architectures, where services are connected via interfaces, and then integrated to form the system. This relates closely to our context of concurrently-executing components in a component-based software architecture.

Canfora et al. further consider reverse engineering to be supportive of *software comprehension*, one of the activities in the software development process, and especially in the software maintenance process. Program and software comprehension are a separate field of study. Janet Siegmund examines this field, by looking at its past, present and future [114]. While her work concentrates on how program comprehension can be measured, one of the conclusions is that research should not focus solely on the source code, but also on getting an overview of large software, including the relationship between components, and their interactions. This fits well with the scope of our work.

Cornelissen et al. provide a systematic survey of program comprehension [37], looking specifically at dynamic analysis techniques (which we discuss below). They too identify that the software needs to be sufficiently understood, before it can be properly changed. One of the sub-fields that they consider, concerns the study of behavioral aspects, which is also the focus of our work. Cornelissen et al. further identify a lack of attention for understanding distributed systems, which relates closely to our context of component-based software systems, as well as for legacy systems, which we consider in some of our case studies.

Another sub-field that is discussed by Cornelissen et al. is *feature analysis*, and more specifically the activity of *feature location*. The goal of that activity is to find out where in the software a certain functionality is implemented. Translated to our context, this could for instance be the identification of where certain messages are communicated, or from where they originate. Dit et al. survey the field of feature location [43], relating the activity strongly to software evolution. They create a taxonomy of analysis techniques, which closely relates to the classes of techniques that we discuss below.

While Chikofsky and Cross, Canfora et al., and Cornelissen et al., all mention software components in the context of their specific fields, Vale et al. specifically survey the field of component-based software engineering [124]. They identify current component specifications as insufficient for understanding their behavior, and their interactions. They advocate for more attention for dynamic aspects of component specifications, rather than only focusing on static aspects, such as the system structure and interface signatures. Our work is specifically aimed at such dynamic aspects, namely understanding and evolving the software behavior in terms of the interactions between components.

Classes of model inference techniques

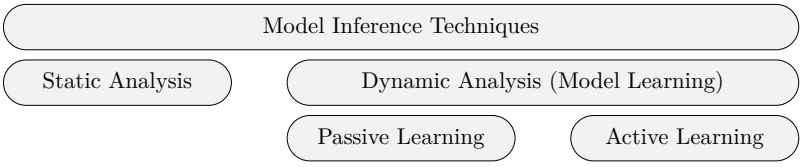
As already mentioned, various techniques exist to infer software behavior models. Figure 1.3 shows two different classifications of such model inference techniques. One of the classifications is based on the artifacts from which the information is inferred (see Figure 1.3a). Broadly, it allows the techniques to be divided into two classes[2]: *static* and *dynamic* techniques [43, 86, 134].

Static techniques work on artifacts used to define, implement and validate the system, such as source code and test code. They are often called *static analysis* techniques, and aim to extract the software behavior directly from the source code and other artifacts, for instance through control flow analysis [46, 95].

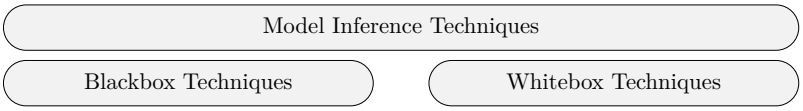
Dynamic analysis techniques work on the executing system or artifacts produced by the executing system, such as execution logs [37]. They are also called *model learning* techniques, and aim to learn the software behavior from observations of the software's behavior [123]. Model learning techniques can be subdivided further into two sub-classes: *passive learning* techniques and *active learning* techniques.

Passive learning techniques infer a model from logs, and aim to generalize beyond the observed behavior of the typically incomplete logs. Examples of passive learning techniques include process mining and state machine learning [1, 60].

[2]Another commonly identified class is that of *historic* techniques, which for instance rely on repository mining [30, 43]. We consider these techniques as less suitable for the scope and aim of our work. They are out-of-scope, and therefore we disregard them.



(a) Classification based on the artifacts from which the information is inferred.



(b) Classification based on the scope from which the information is inferred.

Figure 1.3: Two classifications of model inference techniques.

Active learning techniques, such as *active automata learning*, infer a model by interacting with the system itself, or a part of it. While passive learning techniques typically get as input a predetermined set of logs that cover only part of the behavior, active learning techniques can actively query the system for additional information, to ultimately infer complete models [11, 68, 123].

Model inference techniques can also be classified based on the scope from which the information is inferred (see Figure 1.3b). Broadly, it allows the techniques to be divided into two classes: *black-box* and *white-box* techniques [123]. Black-box techniques consider only the behavior that is observable on the external interfaces of the components (the insides of the components are hidden from view), while white-box techniques (also) consider the internals of the components (you can see inside the box).

Static analysis techniques are white-box techniques, as they have direct access to the source code and other artifacts that implement the components. Dynamic analysis techniques can be both black-box and white-box techniques. For passive learning techniques it depends primarily on the source of the logs; for instance, whether those logs capture the external communications of the components, or their internal function calls. Active automata learning is traditionally a black-box technique [11, 123].

Pros and cons of different classes of techniques

The different techniques each have their pros and cons. Static analysis techniques have access to all the details of the implementation, through the source code. On the one hand, Canfora et al. conclude that static analysis techniques are often reasonably fast, precise, and cheap [30]. On the other hand however, they do require dedicated support for all the different programming languages and language features, that are used in the implementations of the components of the system. Furthermore, static analysis can also be imprecise, and it is especially difficult to extract sequences of exchanged messages, which is our aim. As exact solutions are often computationally intensive, typically abstractions and heuristics are used [18, 46, 95, 113]. The software behavior may then be over-approximated, leading to models that allow behavior that is not possible in the actual system.

Dynamic techniques can be used as *black-box* techniques. This fits well with component-based systems, where the software architecture typically prescribes the use of a common communication middleware that is used by all components. Logs with the communications at the middleware level are then independent of the implementation language used within the components, and active learning can query the component in a common way by using the middleware, regardless of the implementation language.

Dynamic techniques do also come with their own challenges. The main challenge for passive learning is to find proper generalizations, that generalize the behavior beyond what was observed. This typically involves a balance between inferring a model that is as complete as possible and avoiding over-approximations [60]. Some of the main challenges for active learning are scalability and ease of use [13, 44, 68, 111]. The scalability challenges stem among others from the blackbox nature of the technique. Many queries are needed to obtain a complete model, as from the outside it is unclear what queries will discover not-yet-known behavior.

Since scalability is an issue, active learning is typically applied to the individual components of a system, rather than to the entire system. Active learning can also be difficult to use, since (the part of) the system that is to be queried must be isolated and connected to the active learning tool, which often involves significant effort.

Scoping our research

Given the broad range of techniques, we scope our work by focusing on black-box model learning techniques. These techniques allow us to infer the interactions between components, regardless of the many different programming languages that are used to implement the components of the system. We focus in particular on techniques to infer state machines, as state machine models are broadly used in academia and industry alike, including to represent software behavior in general, and software interface protocols in particular. They are also already in use at our industrial partner ASML, easing adoption.

Passive and active model learning techniques, such as state machine learning and active automata learning, can be combined to exploit their complementary strengths [67, 130]. Active learning can ask additional queries to fill the gaps in incomplete models obtained through passive learning. Conversely, using execution logs and models obtained through passive learning can help to reduce the scalability concerns for active learning. Exploiting the complementary strengths of the techniques reduces some of the challenges in applying them, but does not completely eliminate them. For instance, combining passive and active learning does not help to make setting up active learning any easier. And scalability is still a concern, especially for large systems with many components [13].

Hence, it is still of interest to explore how to effectively apply these techniques, and use them to automatically infer the software behavior of large component-based systems.

Sub-dividing RQ-1

We choose to separately address some of the biggest challenges in applying these techniques for large component-based systems. We therefore sub-divide RQ-1 into two smaller research questions.

For active automata learning, in this thesis we focus on a major practical challenge in applying it: the active learning tool needs to be connected to the software for which it aims to learn the behavior. In a component-based system, this involves isolating each component and adding a suitable interface for the learning tool to query the component. Most applications of active automata learning in the literature use ad-hoc solutions [35, 44, 109]. Systematic approaches are rare, but do exist, for instance for web services [94]. However, such a systematic approach is missing for component-based systems operating under the client/server paradigm, making the application of active automata learning laborious for such systems. Hence, we formulate research question RQ-1a:

RQ-1a How can we efficiently set up and apply active automata learning, to automatically infer the software communication behavior, for large component-based systems?

For state machine learning, in this thesis we focus on the major challenge to find proper generalizations, that generalize the behavior beyond what was observed. Existing algorithms from the literature generally use heuristics, which are often based on the input observations (e.g., considering only the last n events [60]) or the resulting model (e.g., limiting the number of resulting states [16]). These heuristics may then be hard to configure properly for engineers in industry. They are often not so familiar with the details of the approaches, and therefore find it difficult to gauge the impact of the heuristics on the learning result, especially for heuristics that do not directly relate to system properties. Existing approaches also do not make use of the unique properties of component-based systems, such as the component structure, the services the components provide, and the (a)synchronous communications between the components. This also applies to modern approaches specifically designed for component-based systems [113]. As a result, there is no guarantee that the inferred models are good approximations of the software communication behavior. In particular, over-generalization beyond the behavior of the actual system is not ruled out, decreasing trust from engineers. Ideally, the models should be as complete as possible, while completely avoiding over-approximations. Hence, we formulate research question RQ-1b:

RQ-1b How can we apply state machine learning, such that it automatically infers good approximations of the software communication behavior, for large component-based systems?

1.3.2 RQ-2: Software change impact

For research question RQ-2, the aim is to efficiently obtain a complete overview of the system-wide impact on the communication behavior caused by software changes, for large component-based systems. In Section 1.2.2 we discussed that engineers must understand the changes they make, as well as any other changes caused by those changes. We therefore aim to assist them in obtaining this overview, by determining all behavioral changes (both for their own component as well as the rest of the system), and presenting these changes in such a way that they can be understood and the impact of the changes becomes clear.

Let us first consider the former, determining all behavioral changes. We could imagine various ways to accomplish this task, such as using the source code, or the documentation. However, we already concluded that understanding the software behavior from such artifacts is either impossible or too difficult. Hence, we study software behavior model inference for RQ-1. Similarly, we consider determining behavioral differences from such artifacts to be practically infeasible. Instead, we rely on the models we infer through model inference, as they contain the relevant information about the communications between the software components, for the entire system. We use these models as input for *model comparison*. For instance, we may have two software versions, one just before a change, and the other just after that change. Then we can infer models of the behavior of both software versions, and we can compare these models to determine all changes in behavior between the software versions. This includes the intended changes, as well as any unintended regressions.

Let us then consider the latter, presenting the changes in such a way that they can be understood and the impact of the changes becomes clear. Given that we consider large component-based systems, there may be many changes, especially in case of ripple effects (as discussed in Section 1.2.2). Providing a single large overview with many changes would be daunting for engineers. We therefore choose a different approach, and present the differences at multiple levels of abstraction. Engineers can then step-by-step zoom in, to explore the differences in more-and-more detail, without losing the overview. Besides that, we also aim to minimize the number of (irrelevant) differences, such that engineers can more efficiently and effectively inspect the relevant differences. This way, engineers can get a good overview of all the behavioral changes, and assess the impact of those changes. It enables them to gain confidence that the changes they made are correctly implemented, and it can help them to prevent regressions. This helps them to reduce the risks involved in making changes.

Next, we look at related literature on software evolution, including change impact analysis. Then, we discuss existing techniques for software behavior comparison, and discuss their pros and cons, in relation to our context and the goals of our work. Based on that, we further scope our work by selecting the type of techniques that we investigate in this thesis. Specifically for those techniques, we identify the gaps that we address in this thesis. And we use those gaps to refine research question RQ-2, by further dividing it into two smaller research questions.

Related literature

Software evolution is a well-studied field. About two decades ago, Mens and Tourwé surveyed software refactoring [93]. They conclude that one of the activities in a refactoring process, is to ensure that the refactorings preserve the software behavior. We focus exclusively on the software behavior. However, we consider software changes in a broader sense than only refactorings. Hence, for some use cases, changes in behavior may actually be expected and desired. Mens and Tourwé further mention that changes may propagate, which is a challenge that we explicitly consider and address in our work.

More recently, Václav Rajlich discusses over a decade of research on software evolution and maintenance [106], and includes impact analysis as an explicit step in the process for making software changes. Rajlich concludes that research on impact analysis is often too crude, for instance indicating only whether a module is affected or not, rather than quantifying that impact. In our work, we focus not only on which components interact differently, but also on what exactly has changed in how they communicate.

Around the same time, Li et al. survey change impact analysis techniques [88]. Similar to our work, they explicitly consider ripple effects, side effects, and regressions. In their work, they classify the techniques into multiple perspectives. Particularly relevant for our context is the 'traditional dependency analysis' perspective, a static analysis perspective involving the use of dependency graphs. Also relevant is the 'execution information collection' perspective, a dynamic analysis perspective involving the use of execution information, for instance execution traces. Their analysis shows that the dynamic techniques may be better in terms of accuracy than the traditional static techniques. We similarly focus on the dynamic

perspective, as we compare models obtained through dynamic analysis techniques.

Li et al. further provide an overview of tools that support the various change impact analysis techniques that they surveyed. While not all those dynamic analysis techniques are supported by tools, the ones that do have tool support, support only specific programming languages. As these tools only support C or Java, they are insufficient for our purposes. Mens and Tourwé further conclude that language independence is desired. In our work, we compare software behavior models that capture communications between components, regardless of the programming languages used to implement the components.

In recent work, Nan Yang also considers dynamic analysis techniques, as she systematically reviews the field of log comparison techniques [129]. She includes in her review not only techniques that directly compare execution logs, but also techniques that compare execution logs against models, and models against models. As in our work, these models may be inferred from the execution logs. We look at such techniques in more detail below. Yang further identifies multi-level comparison as an important need for industry. And she notes a lack of model-to-model comparison approaches that support such multi-level comparisons. In our work, we propose exactly such an approach.

Mahmood et al. specifically discuss the evolution of software components, in their survey on component-based software development [89]. However, they discuss only a few works on evolution, which rely on the structure of the system and its documentation, rather than its behavior. In our work, we focus almost exclusively on the software behavior.

More recently, Vale et al. similarly survey the field of component-based software engineering [124]. They discuss evolution mostly in the context of tooling, such as development environments. However, they also conclude that more research is needed on the evolution of component-based software, including for large distributed systems. And that such research must pay attention to component interactions and interface protocols. We focus on that in our work.

Breivold et al. systemically review software evolution, both in the context of software architecture evolution [26], and open-source software [25]. This differs from our work, as we focus on the implementation of the software, rather than its architecture, and we consider mostly industrial applications of component-based software, rather than only open-source applications.

Van Deursen et al. discuss research on the evolution of software built using model-driven software engineering [42]. This differs from our work, as we use and compare models to support the evolution of software, but not necessarily only for software that is developed using model-driven approaches.

Software behavior comparison techniques

As already mentioned, various techniques exist to compare software behavior. In current industrial practice, engineers often compare execution logs to find differences in behavior that hint at regressions. They use for instance readily-available textual comparison tools [20, 131]. These tools present many low-level differences, that are not necessarily relevant. For instance, interleaving of communications may lead to differences in the order of events in an execution log, while the behavior has essentially remained the same [129, 132]. In the literature, there is quite

some work on comparing logs in a smarter way [10, 19, 52, 129]. However, tuning these algorithms is difficult, since their parameters relate to the input observations rather than the software behavior. For instance, the 2KDiff algorithm highlights sequences of length k or less that appear only in one of the two input logs [10]. But k is a property about logs and not about the actual system, making it difficult for engineers to choose a good value for k, or even to understand the impact of choosing different values of k. Hence, these techniques are not being used in industry [131]. We therefore do not use log comparison techniques in our work.

Besides log comparison, also model comparison can be used to compare software behavior, if models are available, or if they can be inferred from the logs [129]. As we infer state machine models, we focus particularly on comparison of state machine models. Figure 1.4 shows a classification of techniques to compare state machine models. Broadly, such techniques can be divided into *behavior-based* techniques and *structure-based* techniques [126]. Behavior-based comparison considers the externally observable behavior of state machines, in terms of sequences of events. It includes *qualitative relations* and *quantitative measures*. Examples of qualitative relations include equivalence relations, such as bisimulation equivalence, and inclusion relations, such as language inclusion [49]. Examples of quantitative measures include precision and recall [116, 126]. Complementary, structure-based comparison considers the overlap of their internal model representations, in terms of their states and transitions [126].

For large component-based systems, consisting of many components, there may also be many changes in the communication behavior. And these changes may be scattered throughout the system (see Section 1.2.2). Various behavior-based and structure-based techniques can help here. For instance, engineers may pair-wise compare the languages of state machine models of different versions of individual components. This way, they can find out which components of the system have different behavior after the change [111]. And for those components that do have different behavior, structural comparison can give them insight into the actual behavioral differences, such as added function calls, or changes in the order of function calls [126].

We have observed in practice that engineers find quantitative measures difficult to interpret, among others because these measures are difficult to relate to the actual software, and because such values are not always directly comparable. Hence, we do not consider them further in this thesis.

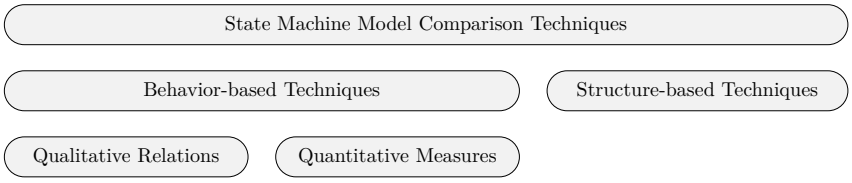


Figure 1.4: Classification of techniques to compare state machine models.

Sub-dividing RQ-2

While various state machine comparison techniques are thus useful, for engineers it is challenging to apply them. To obtain the comparison results, they have to master all of the individual techniques, to know what they are and how to use them. And then they have to apply each of them, one by one, since a single integrated approach is lacking [132]. Furthermore, engineers must apply the techniques to all the models, of all the components of the system, for multiple versions of the software. Clearly, this involves a significant amount of work.

Now assume that the engineers have done this work, by performing hundreds if not thousands of comparisons, and they have the comparison results. Then they face a next challenge, as they have to go through all these results. They have to go through them in detail, one-by-one, to see whether there are any differences at all, and if so, whether it is a relevant difference or not [131]. Given the lack of suitable multi-level overviews of the differences, this again involves a significant amount of work [129].

If instead they would have a multi-level overview, they could at each level see where there are differences, and where there are no differences. If some part of the system has no differences, they do not have to inspect that part in more detail. Thus, they can step-by-step zoom in on only those parts of the system that have differences, to inspect those differences in more detail. Hence, we formulate research question RQ-2a:

RQ-2a How can we combine various state machine comparison techniques into an automated and integrated approach, that allows engineers to efficiently obtain a suitable multi-level overview of all behavioral differences, for large component-based systems?

With suitable multi-level overviews, engineers do not have to spend time on inspecting the parts of the system without behavioral differences, but they still have to inspect the parts of the system that do have differences. For each behavioral difference, they have to see what exactly is the difference, and whether this is a logical consequence of their software change, or an unintended regression. As there may be many components, there may potentially also be many differences to be inspected. It is then essential that there are as few irrelevant differences as possible, and that the relevant differences can easily be inspected. Only then can engineers focus their efforts efficiently, and effectively determine whether their changes introduced any unintended side-effects, to ultimately reduce the risks. Hence, we formulate research question RQ-2b:

RQ-2b How can we present engineers with suitable representations of relevant behavioral differences, for them to efficiently and effectively find any behavioral regressions?

1.3.3 Summary of research questions

Figure 1.5 shows an overview of the research questions of this thesis. Main research question RQ is sub-divided into RQ-1 and RQ-2. RQ-1 is sub-divided into RQ-1a and RQ-1b. RQ-2 is sub-divided into RQ-2a and RQ-2b.

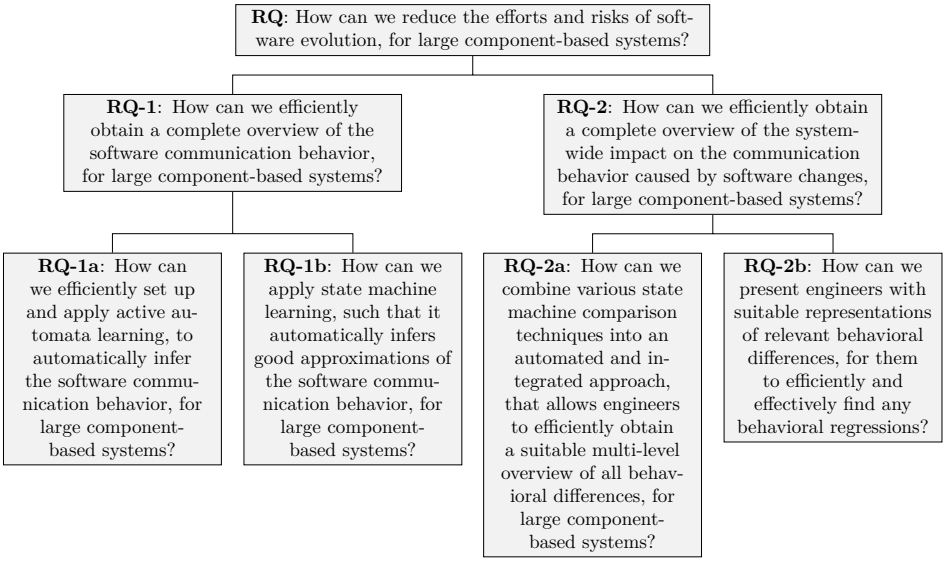


Figure 1.5: Overview of the research questions of this thesis.

1.4 The MIDS methodology

To address the challenges as outlined in the research questions, we develop a methodology that comprises both model inference and model comparison. The methodology is named after the tool that supports it, *MIDS*, which stands for *Model Inference and Differencing Suite* [120]. Figure 1.6 shows the MIDS methodology.

Given a (component-based) system (*System 1*), model inference (the blue parts, in the middle) is used to automatically infer models of its software (*Models 1*). The models may be obtained by *active learning* or by *passive learning*. For active learning, an active learning tool actively queries the system to get observations of its behavior and uses them to produce models. For passive learning, the system is executed and the observed behavior is captured in execution logs, which are used by a passive learning tool to produce models.

As we consider large component-based systems, a single model of the entire system behavior would be much too large and complex. Hence, models are inferred for parts of the system, such as the software components and the functions of their interfaces. The multi-level models provide an overview of the software behavior, as well as detailed information about it, which gives insight into the system's behavior.

Given a second system (*System 2*), models can be obtained in a similar way (*Models 2*). Rather than the second system being a different system, it could also be the same system with for instance different configurations or different software versions. While not visualized in the figure, the methodology allows for any number of systems, configurations, or versions to be considered, and models of their behavior to be inferred.

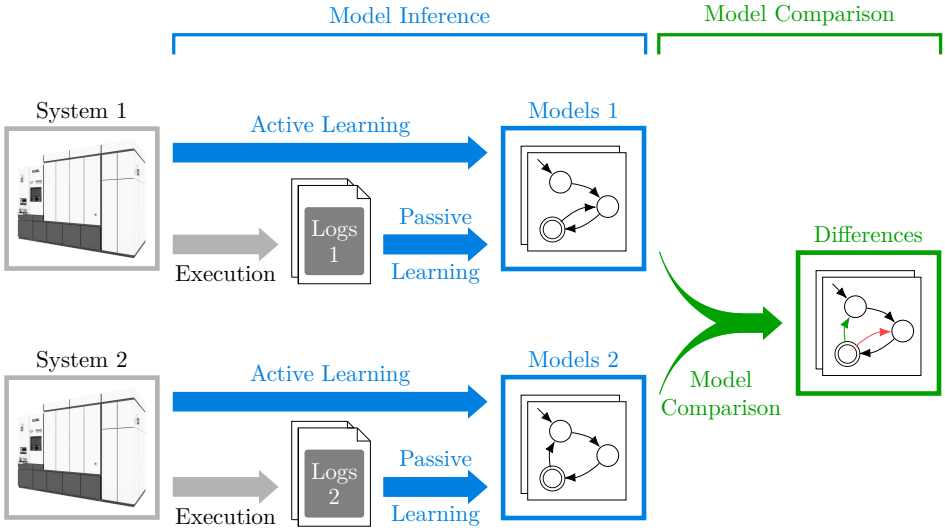


Figure 1.6: An overview of the MIDS methodology, consisting of Model Inference (the blue parts, in the middle) and Model Comparison (the green parts, on the right).

Then, given two or more sets of models, *model comparison* (the green parts, on the right) computes their behavioral differences and outputs difference models (*Differences*). From the difference models, a multi-level report is generated. The report can be inspected, by step-by-step zooming in on parts of the system that have changes. Through inspection, engineers gain insight into the changes that were made, and other changes in behavior caused by them, allowing for change impact analysis. This helps them to identify unexpected differences and regressions, and thereby allows them to increase the confidence in the correctness of the changes and reduce the risks for software evolution.

1.5 Contributions

The MIDS methodology, as described in this thesis, advances the state-of-the-art of model inference and model comparison, to support software evolution for large component-based systems. Specifically, this thesis has five major contributions. The first four major contributions each address one of the detailed research questions. The fifth major contribution involves the MIDS tool that implements the MIDS methodology. It integrates the various other contributions into a single open-source tool, and addresses the overall research question of this thesis.

Note that in this section, we only describe the contributions of this thesis, and how they relate to the research questions. The relations of these contributions to the chapters of this thesis, to already existing publications, and so on, are described in Section 1.6.

1.5.1 Contribution 1: Active automata learning for large component-based systems

Research question RQ-1a states:

RQ-1a How can we efficiently set up and apply active automata learning, to automatically infer the software communication behavior, for large component-based systems?

To address this research question, we develop a systematic approach to connect software components operating under the client/server paradigm to a learning tool. Our general, reusable and configurable framework allows to more quickly produce an active learning setup for such components. The framework deals with the particular challenges that one encounters when interfacing with such components, including their various types of (a)synchronous communications that stem from the components' dual roles as servers to their clients and as clients to their servers. In particular, we systematically derive an interfacing protocol that ensures that certain constraints, such as input-enabledness, single-output-per-input, and finiteness are satisfied, even if the isolated component code does not satisfy these properties. We show the feasibility and effectiveness of our approach by applying it to software components at ASML.

1.5.2 Contribution 2: State machine learning for large component-based systems

Research question RQ-1b states:

RQ-1b How can we apply state machine learning, such that it automatically infers good approximations of the software communication behavior, for large component-based systems?

To address this research question, we develop a novel state machine learning algorithm, *Constructive Model Inference* (*CMI*), that is specifically suited for component-based systems. CMI learns models from software execution logs, which are interpreted using knowledge of the software architecture, its deployment and other characteristics. It does not rely on queries or counter examples, and has no heuristics that would be hard to configure correctly. We instead inject our knowledge of the system's component structure, the services the components provide, and the (a)synchronous communications between the components. This allows us to learn multi-level models that are small enough for engineers to interpret, while accurately capturing the complex system behavior of actual software systems. We demonstrate this by applying the approach to infer models for dozens of components at ASML.

1.5.3 Contribution 3: Behavioral comparison for large component-based systems

Research question RQ-2a states:

RQ-2a How can we combine various state machine comparison techniques into an automated and integrated approach, that allows engineers to efficiently obtain a suitable multi-level overview of all behavioral differences, for large component-based systems?

To address this research question, we develop a novel multi-level approach for behavioral comparison of large component-based systems. The approach integrates multiple existing complementary methods to automatically compare the behavior of state machine models. The comparison results can be inspected at six levels of abstraction, ranging from very high-level differences to very detailed ones. Users are guided through the differences in a step-by-step fashion. At each level the differences are presented with a suitable visualization, that is tailored to allow engineers to zoom in on the parts of the system with differences, wasting no time on the parts without any differences. We evaluate the approach using multiple case studies at ASML, thereby demonstrating that it can be applied to large (sub-)systems, provides engineers insight into the behavioral differences, and allows them to find unintended regressions.

1.5.4 Contribution 4: Improved structural comparison of software behavior

Research question RQ-2b states:

RQ-2b How can we present engineers with suitable representations of relevant behavioral differences, for them to efficiently and effectively find any behavioral regressions?

To address this research question, we focus on level 6 of our multi-level behavioral comparison approach. Through application of our MIDS methodology, we have found that engineers spend most of their time at that level, to interpret the differences, determine whether they are relevant or not, and decide whether they are as expected or constitute regressions. Level 6 is our most detailed level. It shows the behavioral differences based on the results of a structural comparison of state machines. We improve an existing state-of-the-art structural state machine comparison algorithm named LTSDiff [126], by generalizing and extending it to gLTSdiff. gLTSdiff reduces the number of differences represented in the comparison result, which reduces the effort engineers have to spend to interpret the differences. We apply gLTSdiff to several large-scale industrial and open source case studies, to show that it has practical value, can efficiently compute the differences, handles large numbers of input models, and reduces the number of differences.

1.5.5 Contribution 5: An open source tool supporting the MIDS methodology

The overall research question of this thesis states:

RQ How can we reduce the efforts and risks of software evolution, for large component-based systems?

To address this research question, we develop the MIDS tool [120] that implements the MIDS methodology. The MIDS tool provides model inference to infer models of the software. The models give insight into the current communication behavior, which provides a solid basis for making changes to the software. The MIDS tool also provides model comparison to allow engineers to compare software behavior models, zoom in on the relevant differences, determine the impact of software changes, and prevent regressions. Together model inference and model comparison allow to increase confidence that the software changes were made as intended, reducing the risks before these changes are delivered.

The MIDS tool is largely automated. It provides an efficient way to apply the MIDS methodology, reducing the effort that engineers have to spend. The tool is set up in such a way that it can be applied at different companies, which may for instance have execution logs in different formats. The MIDS tool is open source, allowing wide-spread use of the tool and its underlying methodology.

1.6 Thesis outline

The rest of this thesis is structured along the five contributions. Each contribution is described in a separate chapter, as outlined below. Contributions 1 through 4 are based on published papers. To allow reading the chapters in isolation, their individual introduction and conclusion sections are kept. These chapters do contain improvements and extensions compared to the published papers, and the most important differences are described below. The chapter for contribution 5 is a novel contribution for this thesis, and has not been published previously.

The work in this thesis has been conducted in the context of the Transposition applied-research project. This project ran from 2018 through 2022, as a collaboration of TNO-ESI and ASML. I was the scientific and technical lead of the project[3]. The various contributions in this thesis have been collaborations with other members of the Transposition project, its associated PhD and MSc students and their supervisors, as well as industrial contacts at ASML. This is also evident from the list of authors of the published papers on which the chapters are based (see below), and from the acknowledgments sections of those papers. Below, for each chapter based on a published paper, I also indicate my own contribution.

Chapter 2 provides a detailed description of contribution 1, our systematic approach to connect software components operating under the client/server paradigm to a learning tool. The approach is based on a reusable and configurable framework that allows engineers to quickly produce an active learning setup for such components. This chapter is based on the following published paper [55], and is reproduced in this thesis with permission from Springer Nature:

- Dennis Hendriks and Kousar Aslam, *A Systematic Approach for Interfacing Component-Based Software with an Active Automata Learning Tool*, In: Proceedings of the 11th International Symposium On Leveraging Applications

[3]In this thesis, 'we' is used to refer to the author of this thesis, the authors of the publications on which chapters are based (which includes the author of this thesis), or a combination of them together with the reader. Only in this section, 'I' is used to distinguish the individual roles and contributions of the author of this thesis.

of Formal Methods, Verification and Validation (ISoLA), pages 216–236, Springer, 2022, DOI: 10.1007/978-3-031-19756-7_13.

This work was a collaboration with Mladen Skelin, who at the time was a member of the Transposition project. I came up with the ideas for the interfacing approach and interfacing protocol. Mladen did all of the implementation work, and took the lead for the case studies. The work was also a collaboration with Kousar Aslam, a PhD student from Eindhoven University of Technology, who worked closely together with the Transposition project. She included this work in Chapters 6 and 7 of her own PhD thesis [12]. Writing those two chapters was a collaborative effort. For the ISoLA publication, as lead author, I wrote the entire paper from scratch.

The chapter in this thesis differs from the publication. It has mostly small changes, such as textual improvements, style improvements, and some linking to the rest of the thesis. Besides that, the introduction of the chapter is extended.

Chapter 3 provides a detailed description of contribution 2, our Constructive Model Inference (CMI) state machine learning algorithm. CMI is specifically suited for component-based systems by accurately capturing the complex system behavior of such systems. This chapter is based on the following published paper [63], and is reproduced in this thesis with permission from SciTePress:

- Bram Hooimeijer, Marc Geilen, Jan Friso Groote, Dennis Hendriks, and Ramon 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, SciTePress, 2022, DOI: 10.5220/0011145700003266.

This work was a collaboration with Bram Hooimeijer, who conducted his Master's thesis project under the supervision of Jan Friso Groote and Marc Geilen from Eindhoven University of Technology (TU/e), and Ramon Schiffelers from ASML. Building upon earlier work of the Transposition project, and working closely together with the Transposition team, Bram in his project developed the CMI approach, its formalization, and the proofs, and applied the approach to an ASML case study. The published paper is based on Bram's thesis [62]. I took the lead in turning the thesis into the ICSOFT publication, which was a collaboration primarily with Jan Friso and Marc. I among others introduced the 6-step methodology, including the overview figure of the methodology, introduced the running example, and significantly reworked the main methodology section, for instance adding for each step the intuition in the form of an informal description, before discussing the formalization.

The chapter in this thesis differs from the publication. The most notable changes are numerous improved descriptions and extended explanations, more details about the running example, and a completely rewritten section about the analysis of the synchronous variant of the approach.

Chapter 4 provides a detailed description of contribution 3, our approach for behavioral comparison of large component-based systems. The approach produces a multi-level comparison overview, which guides engineers through the differences

in a step by step fashion. It provides them with insight into the behavioral differences, and allows them to find unintended regressions. This chapter is based on the following published paper [56], and is reproduced in this thesis with permission from Springer Nature:

- Dennis Hendriks, Arjan van der Meer, and Wytse Oortwijn, *A Multi-level Methodology for Behavioral Comparison of Software-Intensive Systems*, In: Proceedings of the 27th International Conference on Formal Methods for Industrial Critical Systems (FMICS), pages 226–243, Springer, 2022, DOI: `10.1007/978-3-031-15008-1_15`.

This work was a collaboration within the Transposition project, in particular with Arjan van der Meer, who was a member of the Transposition project until the end of 2022, and Wytse Oortwijn, who joined the project in 2021. Together, we developed the methodology, implemented the tool, and applied it to various case studies, supported by stakeholders and experts from ASML. I took the lead, being the scientific and technical lead of the project. As main author, I wrote nearly the complete FMICS paper.

The chapter in this thesis differs from the publication. The most notable change is an extensive addition that describes the steps involved in the computation of a lattice.

Chapter 5 provides a detailed description of contribution 4, our improved structural state machine comparison algorithm gLTSdiff. gLTSdiff efficiently computes differences for large numbers of state machine models, and reduces the number of differences. This chapter is based on the following published paper [58], and is reproduced in this thesis with permission from IEEE:

- Dennis Hendriks and Wytse Oortwijn, *gLTSdiff: A Generalized Framework for Structural Comparison of Software Behavior*, In: Proceedings of the 26th International Conference on Model Driven Engineering Languages and Systems (MODELS), pages 285–295, IEEE, 2023, DOI: `10.1109/MODELS58315.2023.00025`.

This work was a collaboration within the Transposition project, in particular with Wytse Oortwijn. Together, we developed the approach, implemented the library, and applied it to various case studies. The ASML case studies were a collaboration, primarily with Arjan van der Meer, who was a member of the Transposition project at the time, and João Vieira, who worked at ASML at the time. Being the scientific and technical lead of the project, I was heavily involved in all activities. As main author, I wrote nearly the complete MODELS paper.

The chapter in this thesis differs from the publication. The most notable changes are extensions to the background section, an extended explanation of combining sequences, further elaboration about applying gLTSdiff to multiple inputs, an added example of tangle rewriting, more details about the TLS case study, some updates for new experimental results, and improved explanations of the trade-off evaluation.

Chapter 6 provides a detailed description of contribution 5, our Model Inference and Differencing Suite (MIDS) tool [120]. The tool integrates and automates

the model inference and model differencing approaches as outlined in this thesis, making the methodology broadly available as an open-source tool. This chapter is a novel contribution of this thesis, and has not been published previously.

Chapter 7 concludes this thesis by summarizing the work and reflecting on it. We answer the research questions, discuss some of the limitations of our approaches, and outline future work.

Chapter 2

Systematic software component interfacing for active learning

Cyber-physical systems often employ a component-based software architecture, dividing the system into components that can be developed, tested and deployed independently [119]. Model-Driven Engineering (MDE) places models at the center of attention, allowing for early analysis of a component's software behavior and for implementations to be automatically generated [128]. But gaining such benefits for legacy software requires models, and manual modeling for legacy components is often laborious and error prone due to a lack of understanding of their current behavior, for instance caused by insufficient documentation and the original developers having long since left the company.

To facilitate a cost-effective transition to MDE, model learning can automatically infer first-order models to bootstrap a subsequent manual modeling effort. Passive state machine learning for instance infers models based on execution logs [60] (see also Chapter 3), but the resulting models are often incomplete due to logs covering only parts of the component's behavior. Active automata learning (AAL) on the other hand repeatedly queries the component and has the potential to ultimately infer a model capturing the component's complete behavior. AAL was introduced in Dana Angluin's seminal work on the L* algorithm [11]. A comprehensive body of work extended upon this to, e.g., learn different types of models, improve scalability, and show its practical value [68].

However, practitioners face practical challenges applying AAL to software components of real-world industrial cyber-physical systems. In order for an AAL tool to send queries to a component and gauge its responses, the component must be isolated from its environment and subsequently connected to the learning tool. Existing case studies typically explicitly or implicitly explain their learning setup [35, 44, 109], but establishing such a learning setup is laborious. It can therefore pay off to use a generic setup that can be (re)configured for reuse. By analyzing existing interface descriptions the new configuration can even be automatically generated. This was shown to be effective for web services [94].

But what is lacking is a systematic approach to connect software components operating under the client/server paradigm to a learning tool. Therefore, similar to what is already available for web services, we contribute a general, reusable and configurable framework, to quickly produce an AAL setup, specifically for component-based software with a client/server architecture. Through multiple case studies we show that our semi-automatic approach enables setting up a learning environment to learn (sub-)component behaviors within hours.

A particular challenge when interfacing with such components, is how to deal with their various types of (a)synchronous communications, especially when considering their dual roles as servers to their clients and as clients to their servers. For instance, a reply from a server is an input to the component, but it is only possible after a request from the component itself. Components may thus not be input-enabled, as they may not be able to accept every input in every state, a practical requirement of various AAL algorithms.

If a component is not input-enabled, a learning purpose [3] can be placed between the learner and the component. A learning purpose is essentially a protocol model that is wrapped around the isolated component, such that all communications between the learner and the component go via the protocol model. In our case, the protocol rejects inputs not allowed by the component, such that these inputs lead to a sink state in the inferred model, clearly marking them as not supported by the component. Our protocol forwards any other inputs to the component, to learn the subset of a component's behavior satisfying the protocol. By defining a protocol that allows all the communications that the component supports, a complete model of the component's behavior can in principle still be learned. An essential part of our framework is therefore such an *interfacing protocol* model, a learning purpose that provides a practical but structured way of handling the communications between the AAL tool and the component whose behavior is to be learned. This way, the System Under Learning (SUL), the isolated component code combined with the protocol, is input-enabled. Besides that, the protocol also ensures that the SUL has a single output per input, and represents a finite Mealy machine, even if the isolated component code does not satisfy these properties.

Our main contribution is the systematic derivation of the interfacing protocol for deterministic single-threaded components, that may act as both clients and servers, may perform various types of synchronous and asynchronous communications, and do not make decisions based on data. As an example, we derive such a protocol for the software architecture of ASML, a leading company in developing lithography systems. However, the responsibilities of the interfacing protocol and the way we handle the different types of communication patterns apply to component-based software in general. We therefore believe our work could be used to similarly derive interfacing protocols and set up learning frameworks at other companies with similar software architectures.

The remainder of this chapter is organized as follows. In Section 2.1 we briefly introduce component-based software architectures and AAL. Section 2.2 describes our general AAL framework. Section 2.3 contains our main contribution, as it introduces interfacing protocols and their responsibilities, and describes the systematic derivation of such a protocol for ASML's software architecture. We then apply our approach in practice to infer behavioral models of multiple software

components in Section 2.4, before concluding in Section 2.5.

2.1 Background

Component-based software architectures are often employed by cyber-physical systems to divide the system into *components* that can be developed, tested and deployed independently [119]. A component can act as a *server* offering its functionality via *interfaces* to other components, its *clients*. A component may interact with multiple other components, its *environment*, acting as either client or server to the various components. A single component may be both a server to its clients and a client to its servers. We focus on client-server architectures, where typically clients initiate communication, and servers are idle while not handling requests from their clients. Interactions typically involve calling *functions* from interfaces of other components, e.g., as remote procedure calls. Concurrent components can communicate *synchronously* and *asynchronously* [53]. With synchronous communication a client remains idle while awaiting the server's *response*, while asynchronous communication allows a client to perform other interactions while a server is processing the client's *request*.

Active automata learning (AAL) involves repeatedly querying a component and has the potential to ultimately infer a model that captures the component's complete behavior. The AAL algorithm thus determines which inputs to send to the system to learn more about the system's behavior, and observes the outputs that the system produces. Given our context of component-based software, queries involve function calls. As for instance function calls from clients are naturally inputs, and their corresponding return values are then outputs, we infer Mealy machine models. AAL then involves a *learner* using *membership queries* (MQs), sending sequences of *inputs* (matching, e.g., function calls from clients) to the *System Under Learning* (SUL) and observing the *outputs* (e.g., function return values), using them to construct a *hypothesis* Mealy machine model for the SUL's unknown behavior. An *equivalence oracle* (EO) then either confirms that the hypothesis matches the SUL's behavior or it produces a counterexample. The EO is typically implemented using conformance testing techniques, using *test queries* (TQs) to find differences in behavior between the hypothesis and the SUL. Iteratively, the learner uses the counterexample and further membership queries to refine its hypothesis, and the EO checks this refined hypothesis, until the EO considers a hypothesis correct. See also the left part of Figure 2.1. For a more extensive introduction to AAL, see the literature [11, 68, 123].

2.2 Active automata learning framework

Complex cyber-physical systems may consist of millions of lines of code, spread out over many hundreds of components. Our goal is to infer the externally visible behavior of individual (sub-)components, e.g., to allow replacing the legacy implementation of their business logic. This requires that a component whose behavior is to be learned is isolated from its environment and is subsequently connected to the AAL tool. We assume components are single-threaded, for instance cor-

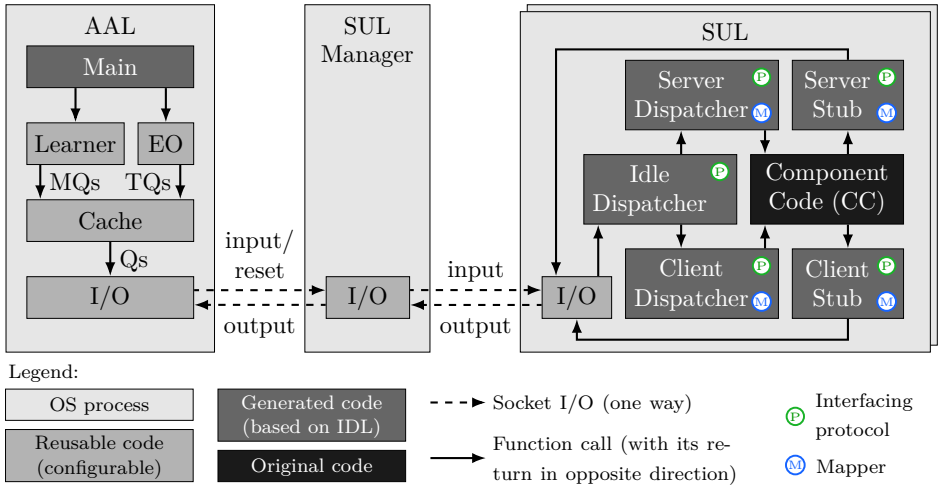


Figure 2.1: General framework to perform AAL on software components operating under the client/server paradigm.

responding to a single Linux thread. Figure 2.1 shows our general framework to perform AAL on software components operating under the client/server paradigm. The outermost boxes represent processes.

At the left is the *AAL* process. It consists of a *Main* function that configures a *Learner* and *EO*, and subsequently iteratively invokes them as described in Section 2.1. The *Learner* and *EO* produce *MQs* and *TQs*, respectively, which are handled identically by a *Cache*. The *Cache* caches and directly answers previously asked queries. For new queries, the *I/O* module sends the *input* symbols to a *SUL Manager* process via sockets, and similarly receives *output* symbols.

The *I/O* module of the *AAL* process sends after each *MQ*/*TQ* a *reset* symbol to the *SUL Manager*, informing it of query completion. The *SUL Manager* uses a new *SUL* instance for every query to ensure that the *SUL* executes inputs from the same initial state. It manages a pool of *SUL* instances and concurrently processes *MQs*/*TQs* and spawns new *SUL* instances for better performance. The *SUL Manager* thus forwards inputs from the *AAL* process to a *SUL* instance, and outputs in the reverse direction. Using multiple processes allows a *SUL* to be implemented in a different programming language than the *AAL* process and eases spawning of new *SUL* instances and killing obsolete ones.

At the core of the *SUL* is the *Component Code* (*CC*) whose behavior is to be inferred. The wrapper code that handles inter-process communications to other components via middleware, dealing with serialization and such, is not considered part of *CC*. Instead, only the code that implements the component's functionality is used, as that is sufficient to learn the component's business logic. Dispatchers and stubs replace the wrapper code, and play the role of the environment of the component, similar to how code is isolated in the field of automated software testing.

When the *I/O* module receives an input, it forwards the input to the *Idle*

Dispatcher, if the *SUL* is idle. A component can be both a server to its clients and a client to its servers. The *Idle Dispatcher* forwards inputs to the *Server Dispatcher* if the *CC* (and thus the SUL) acts as server for requests from clients, or to the *Client Dispatcher* if it acts as client to responses from servers, e.g., due to earlier asynchronous requests to servers. These *Client/Server Dispatchers* act as *mappers* [94] translating input symbols provided as strings to function calls on the *CC*. Once a function returns, its return value is mapped back to an output (string), which via the *Idle Dispatcher* and *I/O* module is provided back to the *SUL Manager* and *AAL* processes. When the *CC* communicates as a client to one of its servers, the call is intercepted in the *Client Stub*. This is also a mapper, forwarding the output symbol corresponding to the call to the *I/O* module. Upon receiving the server reply as *input* from the *I/O* module, the *Client Stub* translates this server reply to a return value to be returned back to the *CC*. Finally, the *Server Stub* similarly intercepts and handles calls from the *CC* (acting as server) to the client. In our work, the mappers do not apply any abstractions to the input and output alphabets.

The final aspect of our framework, the *interfacing protocol*, as implemented in the dispatchers and stubs, is further explained in Section 2.3. The various shades of gray used to color the boxes of Figure 2.1 are explained in Section 2.4.

2.3 Interfacing protocol

In this section, we discuss interfacing protocols and their systematic derivation, using the software architecture of ASML as an example. After briefly explaining the company's software architecture, we describe its communication patterns, and how they map to AAL inputs and outputs, before introducing the interfacing protocol and its responsibilities, and systematically deriving such a protocol from the patterns and the input/output mapping.

2.3.1 Software architecture and communication patterns

The components interact through remote function calls. A call from a client is, by the middleware, placed in the server's message queue. A server is idle while awaiting incoming messages in its `main` function's message processing loop. In this loop, messages are picked up from the queue and processed one by one, non-preemptively. This continues until the queue is empty. The component is then idle again until new messages arrive.

Components communicate with each other using various communication patterns. Clients can invoke functions of their servers in three ways: as *blocking* calls, *request/wait* calls, and *function completion notification (FCN)* calls. Servers can *handle* these requests either *synchronously* or *asynchronously*. The middleware transparently hides these details, as clients are not aware of how servers handle their calls, nor are servers aware of how clients invoke the calls. A client invoke pattern is a partial pattern, as it only concerns the client-side perspective. And so is a server handler pattern, as it only concerns the server-side perspective. Any client invoke pattern can be combined with any server handler pattern to form a complete pattern, as we will see later. We ignore *library* calls, which for the

purpose of this work are identical to synchronously-handled blocking calls. The last two patterns are 'fire-and-forget' type of communications: *triggers* and *events*, which give no assurances of (successful) function execution, e.g., a call may fail without notifying the caller. The trigger and event patterns are complete patterns. Next, we describe the 7 (partial) patterns (3 client requests, 2 server handlers, 2 fire/forget) in more detail using the Message Sequence Charts (MSCs) from Figure 2.2.

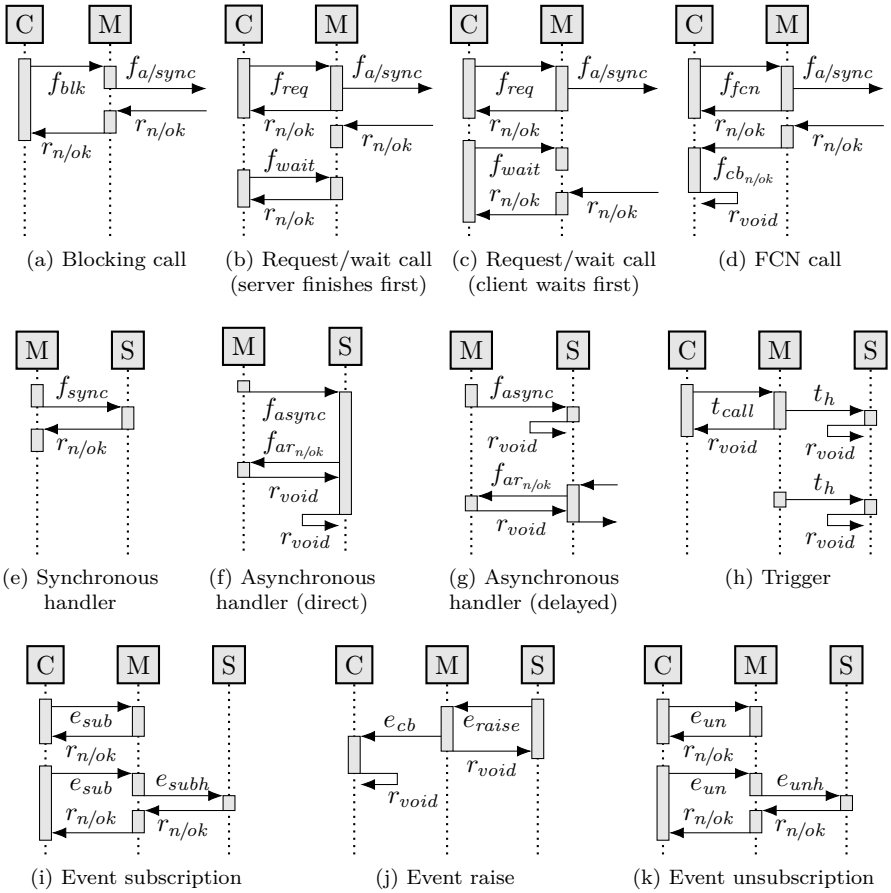
Here f denotes any *function*, t any *trigger* and e any *event*. Collectively, these three are called *methods*. A method is assumed to include the identity of the server that provides it, distinguishing methods from interfaces provided by multiple servers. The company uses a proprietary Interface Description Language (IDL) to define interfaces and their methods. A generator generates, from IDL files, implementation functions to call and handle all defined methods, for various programming languages. We distinguish IDL functions (*i-functions*) from generated functions (*g-functions*) where relevant. For instance, a i-function f from an interface may among others be invoked asynchronously through the request/wait pattern, for which g-functions f_{req} and f_{wait} are generated, as detailed in pattern (ii) below. Any non-void g-function returns a value of type `ASML_RESULT`, an integer result indicating success (0) or failure (non-zero). The company's software architecture rule book states that callers must not use individual error codes, but only OK (0) vs not-OK (non-zero), except for logging, to prevent tight coupling between functions.

(i) Blocking call (Figure 2.2a): A client may invoke an i-function f synchronously as a blocking call (with generated g-function f_{blk}). While awaiting the server's response (return value r_{ok} for successful execution, or r_{nok} for unsuccessful execution), the client is then *blocked* and can not do any internal processing, perform calls, or process messages from its message queue.

(ii) Request/wait call (Figures 2.2b / 2.2c): A client may asynchronously *request* (f_{req}) a server to start executing an i-function f. The client is then free to do other things before explicitly *waiting* (f_{wait}) for the server's response (the second r_{ok} or r_{nok}). If the server has already finished executing the i-function, the middleware has stored the server's response and immediately returns this to the client (**2.2b**). Otherwise, the client is blocked until the server finishes and its response is provided back to the client via the middleware (**2.2c**). With request/wait calls the client is in control of when it is ready to receive the server's response.

(iii) Function completion notification (FCN) call (Figure 2.2d): A client may asynchronously request (f_{fcn}) a server to start executing an i-function f, providing it a callback address (f_{cb}). The client is then free to do other things. Once the server finishes executing f, it provides its response (r_{ok} or r_{nok}) to the middleware, which places both the callback address and the server's response in the client's message queue. Once the client is idle it will process its message queue, eventually processing the server's response using callback g-function f_{cb}, i.e., f_{cb} is called with the queued server response as argument. The client is thus *notified* of the server *completing* the execution of *function* f, as the client requested. With FCN calls the server is in control of when it provides the response.

(iv) Synchronous handler (Figure 2.2e): A server may *synchronously*



(a) Blocking call (b) Request/wait call (server finishes first) (c) Request/wait call (client waits first) (d) FCN call

(e) Synchronous handler (f) Asynchronous handler (direct) (g) Asynchronous handler (delayed) (h) Trigger

(i) Event subscription (j) Event raise (k) Event unsubscription

Lifelines

C	Client
M	Middleware
S	Server

IDL methods

f	IDL function (i-function)
t	IDL trigger
e	IDL event

Messages for generated g-functions

f_{blk}	Blocking call for f
f_{req}	Request call for f
f_{wait}	Wait call for f
f_{fcn}	FCN call for f
f_{cb}	FCN callback for f
$f_{cb_{n/ok}}$	Either $f_{cb}(r_{ok})$ or $f_{cb}(r_{nok})$
f_{sync}	Synchronous handler for f
f_{async}	Asynchronous handler for f
$f_{a/sync}$	Either f_{sync} or f_{async}
f_{ar}	Asynchronous result call for f
$f_{ar_{n/ok}}$	Either $f_{ar}(r_{ok})$ or $f_{ar}(r_{nok})$
t_{call}	Call for t
t_h	Handler for t
e_{sub}	Subscribe call for e
e_{subh}	Subscribe handler for e
e_{raise}	Raise call for e
e_{cb}	Callback for e
e_{un}	Unsubscribe call for e
e_{unh}	Unsubscribe handler for e

Messages for return values

r_{void}	Return value for void g-function
r_{ok}	Return value for successful exec.
r_{nok}	Return value for failed execution
$r_{n/ok}$	Either r_{ok} or r_{nok}

Figure 2.2: MSCs for all (partial) communication patterns.

handle all blocking calls, request/wait calls and FCN calls for an i-function f. Once the handler (f_{sync}) finishes, it immediately returns its response (r_{ok} or r_{nok}).

(v) Asynchronous handler (Figures 2.2f / 2.2g): A server may also *asynchronously handle* all blocking calls, request/wait calls and FCN calls for an i-function f. The asynchronous handler (f_{async}) starts handling the request. In that handler (**2.2f**), or at any later time in any other g-function (**2.2g**), it sends its response to the client by calling an *asynchronous result* g-function, i.e., $f_{ar}(r_{ok})$ or $f_{ar}(r_{nok})$. If it sends the response during the execution of the asynchronous handler (**2.2f**), it effectively acts as a synchronous handler. If on the other hand it sends the response at a later time, the request is truly handled asynchronously, allowing other processing in between (**2.2g**).

A client call pattern (**i** − **iii**) and server handler pattern (**iv** − **v**) are to be combined to form a complete pattern, with a client, middleware and a server.

(vi) Trigger (Figure 2.2h): A client may *trigger* a server (t_{call}), for a trigger t. The server handles (t_h) the trigger without responding back to the client (r_{void}). A server may also be triggered directly by the middleware, e.g., periodically. Since this is more rare, we ignore it in this chapter.

(vii) Event (Figures 2.2i − 2.2k): A client may *subscribe* (e_{sub}) to a specific *event* e of one of its servers (**2.2i**), providing a callback address (e_{cb}). The middleware stores the subscription. A server may optionally have a *subscription handler* (e_{subh}) to be notified of subscriptions. A server may *raise* (e_{raise}) an event (**2.2j**), which leads to callback g-functions (e_{cb}) being invoked (akin to FCN callbacks) on all clients subscribed to that server for the specific event. Clients may at any time *unsubscribe* (e_{un}) from events (**2.2k**), again optionally notifying the server (e_{unh}).

2.3.2 Mapping communication patterns to inputs/outputs

For each of the 7 (partial) communication patterns the various calls to g-functions and their replies can be mapped to inputs and outputs for AAL. Table 2.1 shows this mapping. It is constructed by considering the role of the *CC* (and thus of the SUL), as client and/or server, for each pattern from Figure 2.2. As our goal is to infer a component's functional behavior, it is isolated from its environment, 'cutting off' (ignoring) the middleware. Only the incoming and outgoing messages from clients and servers are considered (C and S lifelines in the MSCs). Messages from a client to the SUL (acting as server) are inputs. Reverse communications are outputs. Conversely, messages from the SUL (acting as client) to a server are outputs. Reverse communications are inputs. The role of the SUL, as client or server, thus inverts whether its incoming and outgoing messages are inputs or outputs.

For instance, in Figure 2.2a the SUL can only act as client. The outgoing f_{blk} message to a server is then an output, and the incoming r_{ok} or r_{nok} message from a server is an input.

2.3.3 The interfacing protocol and its responsibilities

When applying AAL in practice, often several preconditions must hold, e.g., when using LearnLib [70] to learn Mealy machines, the SUL must be input-enabled. If

Communication Pattern	Role of SUL	Message	Input/Output
(i) Blocking call	Client	f_{blk}	output
		r_{ok} / r_{nok}	input
(ii) Request/wait call	Client	f_{req}	output
		r_{ok} / r_{nok}	input
		f_{wait}	output
		r_{ok} / r_{nok}	input
(iii) FCN call	Client	f_{fcn}	output
		r_{ok} / r_{nok}	input
		$f_{cb_{ok}}$ / $f_{cb_{nok}}$	input
		r_{void}	output
(iv) Synchronous handler	Server	f_{sync}	input
		r_{ok} / r_{nok}	output
(v) Asynchronous handler	Server	f_{async}	input
		r_{void}	output
		$f_{ar_{ok}}$ / $f_{ar_{nok}}$	output
		r_{void}	input
(vi) Trigger	Client	t_{call}	output
		r_{void}	input
	Server	t_h	input
		r_{void}	output
(vii) Event	Client	e_{sub}	output
		r_{ok} / r_{nok}	input
		e_{cb}	input
		r_{void}	output
		e_{un}	output
		r_{ok} / r_{nok}	input
	Server	e_{subh}	input
		r_{ok} / r_{nok}	output
		e_{raise}	output
		r_{void}	input
		e_{unh}	input
		r_{ok} / r_{nok}	output

Table 2.1: Communication pattern messages from Figure 2.2 mapped to AAL inputs and outputs, for the SUL acting as client or server.

the *CC* does not satisfy such conditions, the interfacing protocol ensures that the SUL does satisfy them. More precisely, the protocol is implemented in the SUL's dispatchers and stubs. With the *CC*, dispatchers, and stubs all being part of the SUL, the SUL as a whole then does satisfy the conditions (see Figure 2.1). All communications between the learner and the *CC* then adhere the protocol, as the dispatchers and stubs for instance reject any invalid inputs. Here, we discuss the protocol's three responsibilities. The next section explains *how* it satisfies them.

(a) Input-enabled: For some learning tools/algorithms, the SUL must be input-enabled for the learner to query every input on the SUL for every state. This condition does not always hold, e.g., for FCN callbacks (Figure 2.2d). Along with an FCN call (f_{fcn}) the SUL provides a callback address (f_{cb}). In the real system, the middleware places that callback in the client's message queue upon

receiving the server's reply. For AAL, the client dispatcher invokes it directly. An FCN callback to the SUL is an input, which is thus only possible after an FCN call by the SUL. Without the call, the callback address is unknown and it can not be invoked on the *CC*. The SUL is then not input-enabled. The interfacing protocol detects such invalid inputs, for which it can not rely on the *CC*. Instead, the dispatchers and stubs that implement the protocol directly reply to the learner, making the SUL as a whole input-enabled. As only impossible inputs are rejected, the complete *CC* behavior can still be learned for all valid inputs.

(b) Single input, single output: Inferring Mealy machines with AAL requires that each input produces a single output. That is, an input should not lead to zero or multiple outputs, but to exactly one output. The interfacing protocol is designed to always alternate inputs and outputs. By matching g-function calls and their returns, this is a natural fit. It therefore does not prevent learning the full externally-observable behavior of *CC*. As the components we use in our work do not have timeouts, we do not consider them in this chapter.

(c) Finite learning result: Certain component behavior can not be captured as a finite Mealy machine, e.g., for multiple concurrent executions of an asynchronous handler (Figure 2.2g). With a the handler's start (f_{async}), b its end (r_{void}), c a later successful asynchronous result call ($f_{ar_{ok}}$), and d its end (r_{void}), this may lead to sequences '*...ab...cd...*', '*...abab....cdcd...*', etc, and in general '$...(ab)^n...(cd)^n...$'. Unlike a pushdown automaton, a Mealy machine can not capture this in a finite manner. It would contain infinitely many paths, one for each value of $n \geq 0$. AAL would have to discover each path to infer a complete model, which would never terminate. The interfacing protocol can restrict such concurrent executions ($n \leq m$, for some chosen m) to ensure that AAL terminates, at the expense of not learning the full behavior. Not only are higher concurrency variants then absent ($n > m$), any differences in behavior resulting from them would also be absent, e.g., the component's behavior could be different for $n = m+1$ than for any $n \leq m$, but this would not be in the inferred model. However, the resulting models can still be valuable in practice, as the automatically inferred first-order models can be used to bootstrap a subsequent manual modeling effort.

The interfacing protocol is implemented in the dispatchers and stubs, which act as a small wrapper around *CC*, addressing these three responsibilities. Neither ensuring input-enabledness, nor 'single input, single output', prevents learning the full externally-observable behavior of *CC*. Using interaction limit $m = \infty$, the full behavior of a *CC* can thus be learned, assuming *CC* has a finite Mealy machine representation. Otherwise, by restricting m, a subset of the *CC*'s possible behaviors can be queried. As a result of this, the inferred models may be under-approximations of the *CC*'s behavior, over-approximations of it, or a mix of under-approximations and over-approximations for different parts of its behavior. Assuming an AAL algorithm is used that ensures that learned models are *minimal*, our approach in no way impacts that guarantee. We do not consider the condition that the SUL may not exhibit *non-determinism* (different outputs for the same input), as for the components we use in our work we have not encountered any non-determinism.

2.3.4 Systematic derivation of the interfacing protocol

The interfacing protocol provides a structured way of handling the various communication patterns, ensuring that the SUL is input-enabled, provides a single output per input, and represents a finite Mealy machine, even if the isolated *CC* does not satisfy these properties. We derive such an interfacing protocol in a systematic manner, still using the same example software architecture.

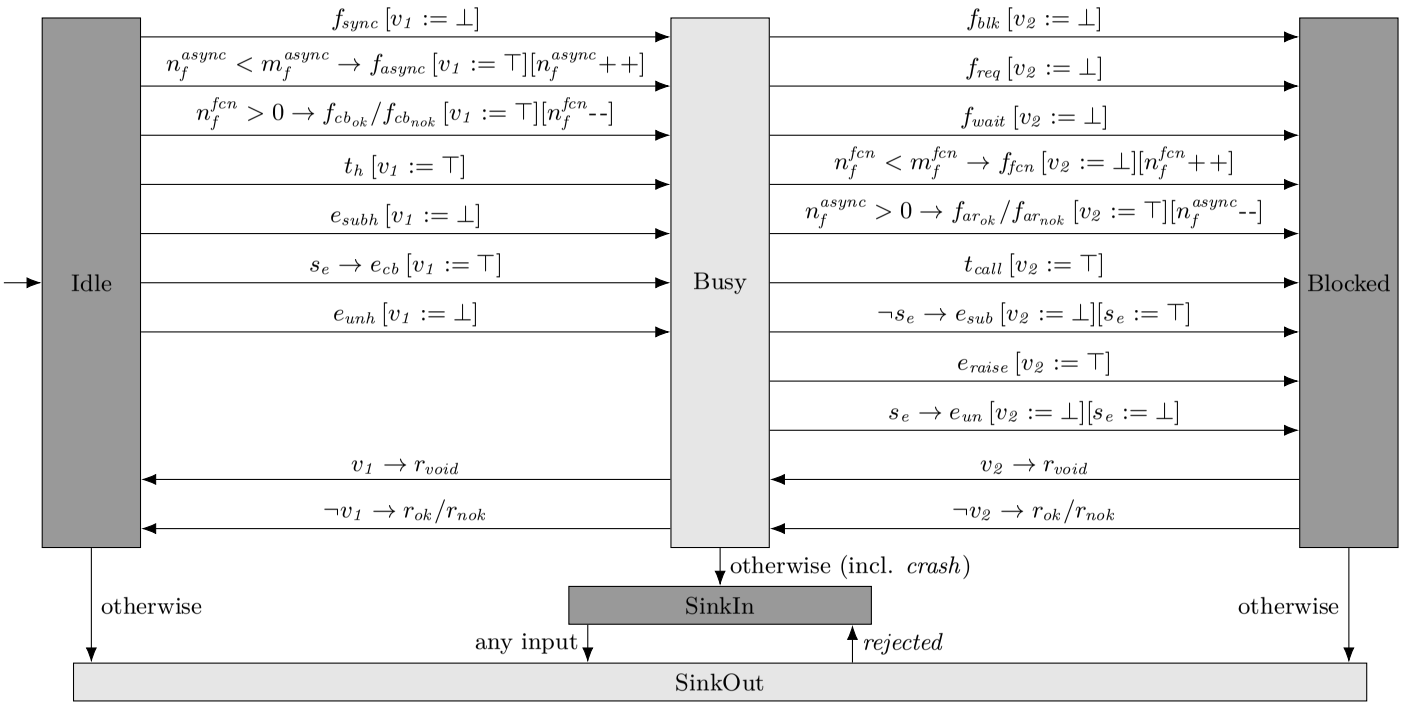
Figure 2.3 shows the interfacing protocol as an Extended Finite State Machine (EFSM). As we infer Mealy machines, each protocol input and following output matches a single Mealy machine transition. We use the syntax $g \rightarrow m\,[u]$ on the edges of the protocol, with g a guard, m a message and u an update. The guard is optional. m_1/m_2 indicates either message m_1 or message m_2. There may be multiple updates, each between square brackets. $x{+}{+}$ is short for the update $x := x + 1$, and x-- is short for the update $x := x - 1$. $\top$ indicates value *true* and $\bot$ indicates values *false*.

The protocol starts in state *Idle* as the *CC* is initially idle, awaiting a call. The *Idle* state is an input state from the *SUL*'s perspective, shown in dark gray in the figure. Upon receiving a *SUL* input, the g-function matching that input is called on the *CC*. The protocol then transitions to the *Busy* state, which is an output state from the the *SUL*'s perspective, denoted light gray in the figure. Here the *CC* continues executing the g-function. Upon its return, the protocol transitions back to the *Idle* state. Alternatively, while *Busy*, it may communicate to one of its servers and go to the *Blocked* state. There it is blocked until the call to the server returns, going back to the *Busy* state to continue the still-in-progress g-function execution. For simplicity, we do not consider calls of a component to its own provided interface. Next, we consider all 7 communication patterns from Figure 2.2, with their associated inputs/outputs from Table 2.1, one at a time. As we gradually build up the interfacing protocol, we consider the patterns from Figure 2.2 in a slightly different order.

(iv) Synchronous handler (Figure 2.2e): An idle SUL, acting as a server, can synchronously handle a call from one of its clients. Upon receiving f_{sync} for some i-function f as an input in the *Idle* state the protocol invokes the corresponding handler (g-function) on the *CC*. It also transitions to the *Busy* state, as the *CC* is then busy executing. Variable v_1 is updated to indicate the in-progress handler is not a void function (update $[v_1 := \bot]$). When the handler returns, depending on its return value (zero for successful execution, non-zero otherwise) the protocol produces an output (r_{ok} for zero, r_{nok} otherwise) and transitions back to the *Idle* state.

(i) Blocking call (Figure 2.2a): The *CC*, while it is executing (state *Busy*), may execute a blocking call to one of its servers. If the *Client Stub* receives a blocking call for an i-function f, it maps that to output f_{blk} and transitions to *Blocked*. A blocking call is a non-void g-function ($[v_2 := \bot]$). In *Blocked* the SUL is blocked while waiting until it receives r_{ok} or r_{nok} as input. It then returns from the blocking call back to the *CC*, with return value 0 (for r_{ok}) or 1 (for r_{nok}), and transitions back to *Busy*.

(ii) Request/wait call (Figures 2.2b / 2.2c): Similar to blocking calls, the *CC* may perform request/wait calls (f_{req} and f_{wait}), going from *Busy* to *Blocked* and back (r_{ok} or r_{nok}).



f	IDL function (i-function)	v_1	Variable	$\top$ if the current/last in-progress call for state *Busy* returns r_{void}	$\bot$ if it returns r_{ok} or r_{nok}
		v_2	Variable	$\top$ if the current/last in-progress call for state *Blocked* returns r_{void}	$\bot$ if it returns r_{ok} or r_{nok}
t	IDL trigger	s_e	Variable	$\top$ if SUL is subscribed to event e	$\bot$ otherwise
e	IDL event	n_f^{async}	Variable	Number of f_{async} calls still requiring a matching $f_{ar_{ok}}$ or $f_{ar_{nok}}$	$0 \leq n_f^{async} \leq m_f^{async}$
		n_f^{fcn}	Variable	Number of f_{fcn} calls still requiring a matching $f_{cb_{ok}}$ or $f_{cb_{nok}}$	$0 \leq n_f^{fcn} \leq m_f^{fcn}$

Figure 2.3: Interfacing protocol EFSM for ASML's software architecture and communication patterns.

(iii) FCN call (Figure 2.2d): The *CC* may also invoke an FCN call (f_{fcn}), going from *Busy* to *Blocked* and back (r_{ok} or r_{nok}). n_f^{fcn} is incremented by one ($[n_f^{fcn}{+}{+}]$) to indicate the FCN callback corresponding to this FCN call has not yet been handled. At a later time, when the SUL is *Idle*, it may handle the FCN callback, i.e., $f_{cb_{ok}}$ in case of success or $f_{cb_{nok}}$ upon failure of the FCN call. An FCN callback is a void g-function ($[v_1 := \top]$), and n_f^{fcn} is then decreased by one ($[n_f^{fcn}{-}{-}]$). For any i-function f, the protocol restricts the number of concurrently outstanding FCN calls (n_f^{fcn}) to at most m_f^{fcn}. Its callback ($f_{cb_{ok}}$ or $f_{cb_{nok}}$) is only possible if there is an outstanding FCN call (guard '$n_f^{fcn} > 0 \rightarrow$'). For simplicity, we ignore the use of FCN call timeouts.

(v) Asynchronous handler (Figures 2.2f / 2.2g): Similar to synchronous handlers, the SUL may asynchronously handle calls from its clients (f_{async}). Such handlers are void g-functions ($[v_1 := \top]$). The *CC* may, during that handler or at any later time that it is *Busy*, invoke an asynchronous result g-function for this asynchronous handler ($f_{ar_{ok}}$ or $f_{ar_{nok}}$), which returns r_{void}. Variable $0 \leq n_f^{async} \leq m_f^{async}$ keep track of and restricts the number of concurrently outstanding asynchronous handler calls for i-function f.

(vi) Trigger (Figure 2.2h): While *Busy*, the SUL can trigger a server (t_{call}), returning void ($[v_2 := \top]$). While *Idle*, the SUL can handle a trigger from a client (t_h), also returning void ($[v_1 := \top]$).

(vii) Event (Figures 2.2i – 2.2k): While *Busy*, a SUL may subscribe to an event of a server (e_{sub}), after which it is subscribed ($[s_e := \top]$). It can only do so if not yet subscribed to that event of that server ($\neg s_e \rightarrow$). Similarly, it may unsubscribe (e_{un}) if already subscribed ($s_e \rightarrow$) and is then no longer subscribed ($[s_e := \bot]$). While *Idle* and subscribed ($s_e \rightarrow$), it may process event callbacks (e_{cb}). Reversely, acting as a server to its clients, it may execute event (un)subscription handlers (e_{subh} and e_{unh}) while *Idle*, and raise events (e_{raise}) while *Busy*. For simplicity, we ignore the rare use of re-subscriptions.

States *Idle*, *Busy* and *Blocked*, and the transitions between them, support all 7 communication patterns, i.e., allow the interaction patterns modeled as MSCs in Figure 2.2. Next, we explain how the protocol satisfies its responsibilities.

(a) Input-enabled: Some inputs are impossible in certain input states. For instance, a t_h input is possible in state *Idle*, but not in state *Blocked*. Also, $f_{cb_{ok}}$ is only allowed in *Idle* if $n_f^{fcn} > 0$ holds. For all impossible inputs in input states, the interfacing protocol transitions to a sink state, where it keeps producing *rejected* outputs. That is, for invalid inputs it goes to the *SinkOut* output state. There it produces output *rejected*, goes to input sink state *SinkIn*, where it accepts any input, goes to *SinkOut*, produces *rejected* as output, etc. This turns a non-input-enabled *CC* into an input-enabled *SUL*, while preserving all its original externally-observable communication behavior.

(b) Single input, single output: Each of the five protocol states is either an input state (dark gray) or output state (light gray). Input states have only outgoing transitions for inputs, and output states only for outputs. Transitions for inputs go to output states, while output transitions lead to input states. It intuitively matches the duality of g-function call starts and their returns. If the *CC* crashes in state *Busy*, the protocol produces a single *crash* output symbol and

goes to *SinkIn*. This way the protocol ensures that each input is followed by a single output, and that they alternate. It also remains input-enabled, and still supports all communication patterns to allow inferring the full *CC* behavior.

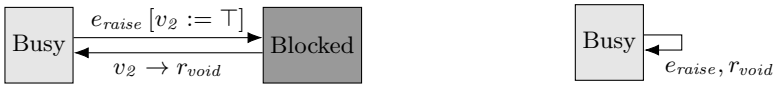
(c) Finite learning result: To ensure that the SUL represents a finite Mealy machine, certain interactions can be limited. For instance, m_f^{async} limits the number of concurrently outstanding asynchronous handlers for i-function f. Starting with $m_f^{async} = \infty$, intermediate hypotheses may reveal it is necessary to restrict m_f^{async}. This ensures a finite SUL and learning result at the expense of potentially missing some component behavior. The protocol restricts both inputs (e.g., f_{async}) and outputs (e.g., $f_{ar_{ok}}$), redirecting them to sink states. The protocol in Figure 2.3 limits only outstanding FCN calls and asynchronous handlers. Theoretically, similar issues could arise for other communication patterns. These can similarly be restricted, but this has been omitted here, to keep the protocol simpler and because they rarely need to be restricted in practice. In particular, request/wait calls are not restricted as they involve only outputs, not inputs, and the company's software architecture rule book, to which all its software must adhere, already allows at most one concurrently outstanding request per i-function f.

The complete behavior of a *CC* can be learned, if it is finitely representable as a Mealy machine, by setting all interaction limits to ∞. Otherwise, by setting interaction limits, a subset of the *CC* behavior can be queried, and we lose the guarantee that the SUL's complete behavior can be learned. We do not provide a formal proof of the correctness of our approach, leaving this as future work.

2.3.5 Interfacing protocol optimization

In the interfacing protocol (Figure 2.3) the *CC* may, while *Busy*, raise an event (e_{raise}). It is then *Blocked* until the event raise g-function returns (r_{void}). The part from Figure 2.3 related to raising events is shown in Figure 2.4a. While in state *Blocked* only one input (r_{void}) is allowed, the learner will try out all inputs, only to find out all of them get rejected, except for r_{void}. This holds any time an event is raised by the *CC*.

This can be optimized as shown in Figure 2.4b. Here the output (e_{raise}) and subsequent input (r_{void}) are combined into a single transition. To preserve the *single input, single output* property of the interface protocol, the e_{raise}, r_{void} transition from output state *Busy* is considered an 'extra output'. Upon executing such a self-loop, the protocol stores the extra outputs until the next 'real' output. It then prefixes the 'real' output with the extra outputs, in the order they were produced. The mapper, being part of the protocol, maps each of them to a string and combines them to form a single output. For instance, for two consecutive



(a) Non-optimized (b) Optimized

Figure 2.4: Optimization of the interfacing protocol, for raising an event.

Mealy transitions i/e_{raise} and r_{void}/o, with i some input and o some output, the optimized result would be a single Mealy transition $i/e_{raise}, r_{void}, o$.

To reverse the optimization after learning, the Mealy transition in the learned model can be split up into multiple Mealy transitions: i/e_{raise} and r_{void}/o, with an intermediate state in between. From the intermediate state, all other inputs are rejected and go to the sink state. The learning result is then the same as it would have been without the optimization.

All void g-function outputs from *Busy* to *Blocked* allow for this optimization, i.e., $f_{ar_{ok}}$, $f_{ar_{nok}}$, t_{call}, and e_{raise}.

2.4 Application

We apply our approach to infer the behavior of two ASML software components: a high-level wafer exposure controller (we name it WEC), and a metrology driver (MD). However, these case studies are not a main contribution of our work, but rather examples to show the feasibility of applying our framework in practice, and to discuss the practicalities that it involves. Therefore, and for reasons of confidentiality, we do not describe them in more detail.

For each component, the AAL framework from Figure 2.1 needs to be instantiated. The following steps were involved:

1. Framework generation: For the component of interest, its interfaces must be identified, and their relevant code (g-functions) collected to use as *CC*. The company's proprietary generator takes IDL files with interface methods and automatically generates g-functions. We extended it to generate the *Main* function, dispatchers and stubs, including code to implement the interfacing protocol and mappers. The three *I/O* modules are hand-written and reusable. The *SUL*'s *I/O* module includes its `main` function, which establishes a socket connection with the *SUL Manager* and waits for inputs to dispatch. While any AAL tool can be used, we opt for the mature AAL tool LearnLib [70], making our *AAL* process Java-based. The *SUL Manager* and *SUL*, including the *CC*, are C-based.

2. Initialization: We manually add initialization code to the *SUL*'s new `main` function, reusing code from the component's original `main` function. The new `main` function replaces the original `main` function.

3. Function parameters: WEC and MD are control components. They pass function call parameter values along, rather than them being used for control decisions (e.g., `if` statement conditions). We therefore mostly ignore function call parameters, rather than learning register automata. Our generator generates default values for function call arguments (0 for *int*, `NULL` for pointers, etc). This may be insufficient, e.g., when the *CC* tries to pass along a field of a `NULL`-valued argument for a `struct`-typed parameter. It was an iterative process to manually adapt these values, where the *CC* and existing test code served as inspiration.

4. Interaction limits: Based on expert knowledge of WEC and MD, we set both interfacing protocol interaction limits: $m_f^{async} = 1$ and $m_f^{fcn} = 1$. Using this low value reduces the size of the *SUL*'s unknown behavior model, for better AAL performance.

5. SUL compilation: We adapt the component's build scripts, excluding any irrelevant code, e.g., its original `main` function and serialization wrappers, and including the new dispatchers, stubs, and *I/O* module, before compiling the SUL.

6. SUL Manager compilation: We configure the SUL pool size to 100, as higher did not help on our platform. We then compile the *SUL Manager* using its generated build script.

7. Learner and EO: Our generated *Main* Java class is by default configured to use TTT [71] as *Learner* and Wp [47] as *EO*. To guarantee learning a complete model, Wp requires n, the number of states of the *SUL*'s unknown behavior model. As we don't know n, we use Wp-incremental, where we guess n (or just start at 0), and iteratively increase the value if needed [12]. Caching is also enabled by default.

8. Input alphabet: The generated *Main* class by default configures the complete input alphabet as derived from the IDL files. It can be reduced to only learn a part of the component's behavior. Considering all provided (to clients) and required (to servers) interfaces, WEC has 591 inputs. It implements 25 distinct workflows. We select five of them, of various complexities, and learn them, including their prerequisite other workflows. For component MD, we keep the complete alphabet with 25 inputs.

9. AAL process compilation: We compile the *AAL* process executable.

10. Perform AAL: Finally, we execute the *AAL* process executable to learn models, repeating earlier steps in case of required changes, e.g., after adapting function call arguments or protocol interaction limits.

Each experiment was executed for 24 hours. For WEC, a dedicated system with 24 CPU cores (Intel Xeon Gold 6126) and 64 GB memory was used. For MD, a readily-available virtualized platform with shared resources was used.

We consider the learning/testing rounds up to and including the last learning round that produced the largest hypothesis, and omit subsequent testing that did not find any more counterexamples. Table 2.2 shows for each (sub-)component (C) the number of inputs (I), the Wp EO n value (Wp-n), the number of Mealy machine states in the model we learned (M-n), the number of equivalence queries (EQs), the number of membership queries (MQs) and membership symbols (MSs), the number of test queries (TQs) and test symbols (TSs), both to the cache (/C) and to the SUL (/S), and the total time in seconds (T).

WEC-1 and WEC-2 are small workflows, without prerequisites. Their largest hypotheses are produced within a few seconds, and no new behavior was found during the many remaining hours of AAL execution. Manual inspection of the component code leads us to conclude they have been learned completely.

WEC-3, WEC-4 and WEC-5 have other workflows as prerequisites. Their largest hypotheses are produced within a few hours. However, they do not accept traces that we manually constructed based on their source code. The traces have 86, 100 and 101 inputs, respectively. Learning thus did not yet find all their behavior. This is to be expected though, given that it is hard for black-box testing to find the exact (combination of) prerequisite sequences to test out of all possible test sequences. And even more so considering that we test breadth-first by incrementally increasing the n value of the Wp EO.

C	WEC-1	WEC-2	WEC-3	WEC-4	WEC-5	MD
I	2	6	25	26	30	25
Wp-n	46	17	66	39	66	916
M-n	50	23	71	55	71	917
EQs	5	4	13	8	13	544
MQs/C	538	967	8,876	7,685	10,644	98,635
MQs/S	52	129	1,462	1,281	1,779	38,300
MSs/C	12,554	12,015	143,335	112,681	171,851	2,122,205
MSs/S	1,015	1,490	23,020	18,266	28,072	854,765
TQs/C	1.59×10^{7}	1,435	1.67×10^{9}	6.57×10^{9}	4.13×10^{9}	9.80×10^{7}
TQs/S	43	3	5,543	3,048	22,689	196,686
TSs/C	4.91×10^{8}	14,074	3.08×10^{10}	1.06×10^{11}	7.65×10^{10}	2.00×10^{9}
TSs/S	1,399	36	88,398	41,046	372,376	4,262,369
T	23	5	2,171	7,569	5,870	62,604

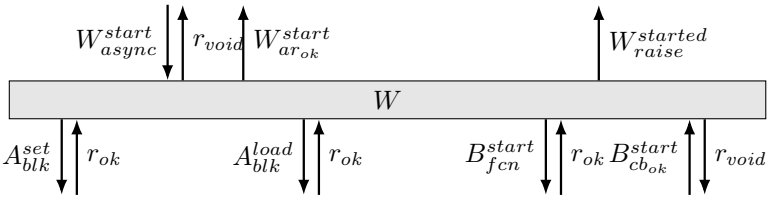
Table 2.2: Case study metrics, per (sub-)component.

For component MD in total 544 hypotheses are constructed in about 17.4 hours. The last hypothesis accepts our manually constructed trace. Acceptance of this trace, and no further counterexample being found for the remaining 6.6 hours, gives us some confidence that we might have found the complete behavior, although we do not have any evidence that we indeed found all behavior.

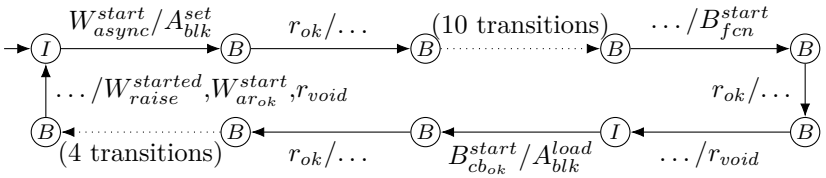
The learned models can be used for various purposes [13]. Here, our goal is to facilitate a cost-effective transition to MDE, concretely to Verum's commercial ASD Suite[1], which is based on their patented ASD technology [28], and is used by ASML as MDE tooling[2]. To exemplify this, Figure 2.5a shows WEC-2 (abbreviated to W) and part of its anonymized context. Figure 2.5b shows a part of the anonymized learned Mealy machine of WEC-2. The sink state and its incoming and outgoing transitions are purposely omitted. Figure 2.5c shows the result of manual conversion of the Mealy machine to a partial ASD design model. The conversion is straightforward: Mealy machine states and transitions correspond one-on-one to states and transitions in the ASD model, where inputs become triggers and outputs become actions. For simplicity, we ignore function parameters. ASD requires for each control component both an interface model and a design model. An interface model can be automatically obtained from the AAL result [13], and then similarly converted to an ASD interface model. From the ASD models, new component code can be automatically generated using the ASD Suite. This can then replace the existing *CC*. All that remains is to update the glue code as needed, and to include the ASD runtime for compilation. If a complete Mealy machine of the *CC* was learned, the newly generated code is then a drop-in replacement, its externally-visible communication behavior being identical to that of (the learned model of) *CC*.

[1] See `https://verum.com/asd`.

[2] The company has since moved to a different but similar MDE tooling solution, which does not affect our approach.



(a) W and some of its communications with its context.



(b) Mealy machine representing a partial learning result. Mealy machine states are labelled with interfacing protocol states: I=Idle, Y=Busy, B=Blocked.

W (W.dm)

W | Data Variables | Service References | Implemented Service | Used Services

SBS | States | State Variables | State Diagram

s1 | s1 (initial state)

	Interface	Event	Guard	Actions	State Variable Updates	Target State
1	**s1 (initial state)**					
4	W	start_async		A:A.set+		s2
10	**s2 (synchronous return state)**					
14	A:A	OK		...		s3
19	**s3 (synchronous return state)**					
...						
109	**s13 (synchronous return state)**					
117	...	...		B:B.start_fcn+		s14
118	**s14 (synchronous return state)**					
123	B:B	OK		...		s15
127	**s15 (synchronous return state)**					
135	...	...		W.VoidReply		s16
136	**s16**					
143	B:B	start_cb_ok		A:A.load+		s17
145	**s17 (synchronous return state)**					
149	A:A	OK		...		s18
154	**s18 (synchronous return state)**					
...						
190	**s22 (synchronous return state)**					
198	...	...		W.started; W.start_ar_ok B:B.VoidReply		s1

(c) Screenshot of a partial ASD design model in ASD Suite. Irrelevant rows are hidden.

Figure 2.5: Partial example of converting learning results to ASD, for WEC-2 (abbreviated 'W'). Naming scheme: W^{start}_{async} is an asynchronous handler for function *start* of W.

2.5 Conclusions and future work

In this chapter, we describe a general AAL framework to learn the external communication behavior of software components with a client-server architecture, filling a practical gap when applying AAL to such components. Our framework includes an interfacing protocol, which ensures that the SUL satisfies various practical preconditions of AAL, even if the isolated component code does not satisfy them. It is future work to infer pushdown automata to prevent having to use interaction limits to ensure finite learning results.

Our main contribution is the systematic way in which we derive the protocol, handling the different types of (a)synchronous communications. We derive, as an example, such a protocol specifically for ASML's software architecture. However, we rely on generic concepts, e.g., function calls and returns, requests and replies, synchronous vs asynchronous calls, and MSCs, that apply to communication patterns of component-based software in general. We therefore expect that our work can be used to similarly derive such protocols and set up learning frameworks at companies with similar software architectures.

We show the feasibility of our approach by applying it to infer the behavior of several ASML (sub-)components. We believe that company engineers should be able to similarly apply our framework, given only a document with detailed instructions, which is future work.

Using generators we automate most of the work to set up an AAL environment. Still, this takes up to a few hours per (sub-)component. It is especially time-consuming to provide sensible function call arguments, to ensure that the SUL does not crash and thus exhibits relevant behavior. It is future work to automate this using white-box techniques, and to infer register automata for components with argument-dependent behavior.

Furthermore, scalability remains a major challenge. Even after hundreds of billions of test symbols, the complete behavior was not learned for some sub-components. There are various techniques that can improve active learning performance, including checkpointing, incremental equivalence queries [66], white-box approaches [68] and incorporating available traces [130]. Integrating them into our framework is also future work.

Still, for some (sub-)components we learned the complete behavior well within a day. This can significantly reduce the time to obtain a model of their behavior, compared to modeling them in a completely manual way. It is future work to automate the conversion to ASD, and further investigate the qualitative and quantitative advantages of our approach compared to manual modeling.

Chapter 3

CMI: State machine learning for component-based software

Model-based software engineering has been shown to be able to cope with the increasing complexity of software [5]. However, for most software, especially *legacy software*, no models exist and constructing models manually is laborious and error-prone. This is especially the case for models that capture the behavior of the software, for instance the order of function calls and returns.

A solution is to infer the models automatically from software execution logs, observations of the system behavior. Model inference has been studied in the fields of *model learning* [60] and *process mining* [1]. Both encompass a vast body of research, and do not focus specifically on software systems. Still, the techniques have been applied to software components in general [2, 59, 113], and specifically to legacy components [21, 83, 111].

There are known fundamental limitations to the capabilities of model inference: Mark Gold proved that accurately generalizing a model beyond observations (logs) is impossible based on observations alone [51]. Accurate learning from logs requires additional information, such as for instance counter examples, i.e., program runs that can never be executed. In practice, heuristics are used, for instance based on the input observations (e.g., considering only the last n events [60]) or the resulting model (e.g., limiting the number of resulting states [16]). These heuristics may then be hard to configure properly for engineers in industry. They are often not so familiar with the details of the approaches, and therefore find it difficult to gauge the impact of the heuristics on the learning result, especially for heuristics that do not directly relate to system properties.

Alternative to using counter examples, active automata learning queries (parts of) the system [11] (see Chapter 2). It guarantees – under certain assumptions – that the inferred models exactly match the behavior of the software implementation. But it suffers from scalability issues [13, 68, 130], limiting the learned models to a few thousand states at most.

We aim to infer models for large industrial systems. We therefore introduce

a novel state machine learning algorithm, Constructive Model Inference (CMI), which learns models from software execution logs, interpreting them using knowledge of the software architecture, its deployment, and other characteristics. It does not rely on queries or counter examples, and has no hard-to-configure heuristics.

We instead inject our knowledge of the system's component structure, and the services each component provides, which it can implement by invoking services provided by other components. After a component executes a service, it returns a response and is ready to again provide its services. This knowledge of the components and their services is essential to cope with the complexity of the industrial systems we deal with. The CMI approach allows us to learn multi-level models that are small enough for engineers to interpret, while capturing the complex system behavior of actual software systems. We apply CMI to dozens of components from ASML's TWINSCAN system. The components have a combined state space that is too large to interpret (i.e., 10^{10} states and beyond). To the best of our knowledge no similar approach exists.

To demonstrate that our CMI approach is adequate, we show that if a component-based software architecture satisfies our assumptions, CMI returns correct results, both in settings with synchronous and asynchronous communication between the concurrent components. Inferred models are correct if they accept at least the input observations, and generalize beyond those observations, but do not over-generalize beyond the behavior of the actual system. The latter is especially important, as models with behavior that the system can never exhibit decrease the trust of engineers in the models. The CMI approach prevents such over-generalization by only generalizing based on knowledge of the system, so only when it is justified. Additionally, a correct model inference approach should ensure that using additional observations only leads to models with more behavior, never less behavior.

While the actual software largely adheres to our structural and behavioral assumptions, there are however parts that do not. We deal with this by analyzing the learned models, e.g., by searching for deadlocks. Part of the CMI approach is a systematic method to add additional knowledge, to exclude from the models any behavior known to not be exhibited by the system.

Inspection of the learned models by experts led to the judgment that the models are very adequate, and provide them the software behavior abstractions that they currently lack [132]. Learning larger models is limited not by the software size, but by the capability of both engineers and computers to analyze those models.

We first present an overview of the approach in Section 3.1, and recall basic definitions in Section 3.2. We then describe and analyze the CMI approach, our main contribution, for synchronously and asynchronously composed component-based systems, in Sections 3.3 and 3.4, respectively. Finally, we outline a method to apply the approach, using a case study at ASML as an example, in Section 3.5, and draw conclusions in Section 3.6.

3.1 Constructive Model Inference overview

We present a high-level outline of the CMI approach before discussing the detailed steps in the following sections. Figure 3.1 illustrates the overall approach and the

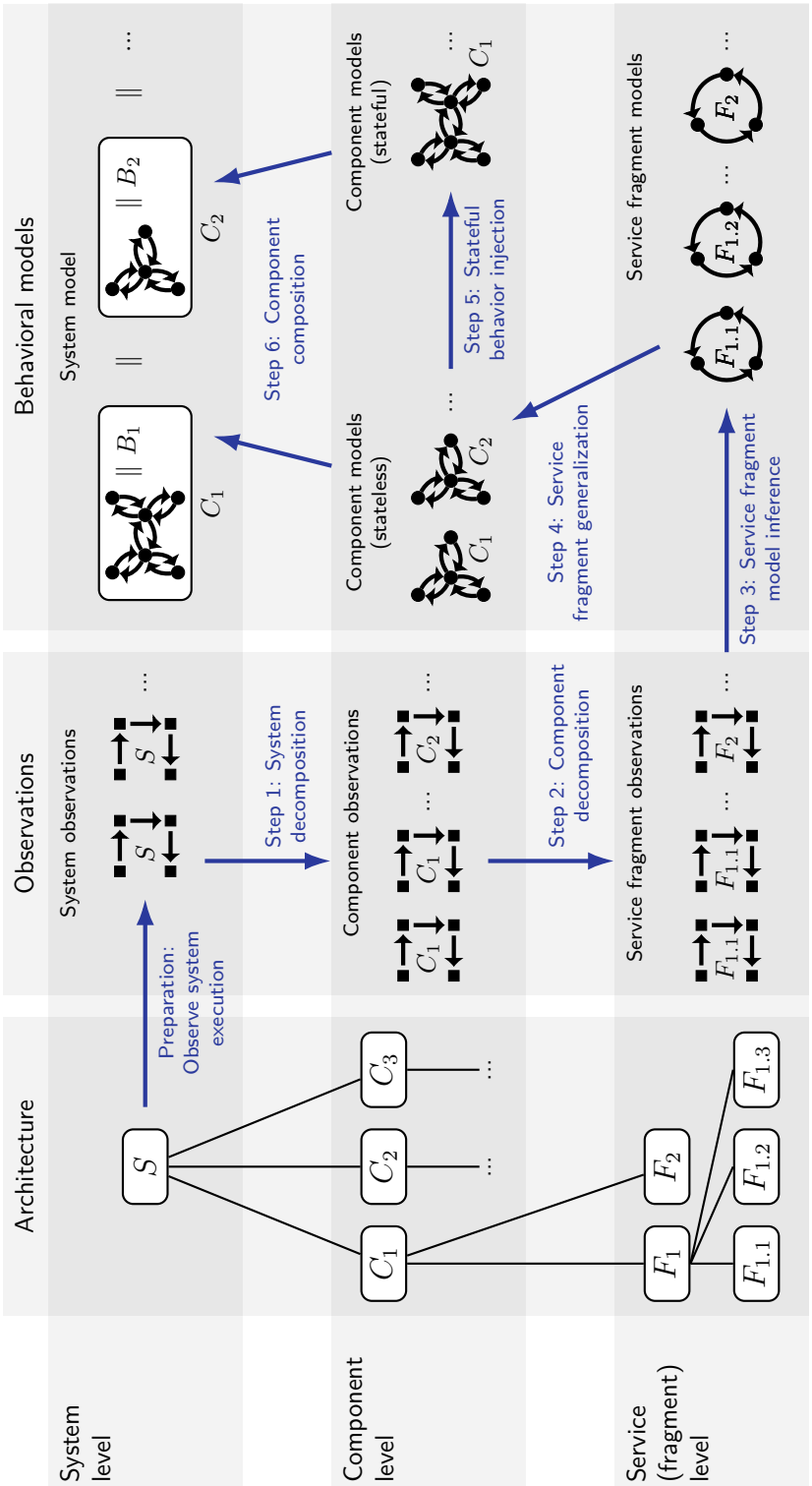


Figure 3.1: Constructive Model Inference (CMI): Overview and its six steps, positioned along abstraction levels (rows) and conceptual views (columns).

steps involved. The left column shows the assumed system architecture. A *system* (S) consists of a known set of *components* (C_1, C_2, C_3) that collaborate, communicate and use each other's services through function calls. Different functions may for instance represent requests to other components, responses from other components, or handlers to handle requests or responses. A component, in turn, consists of the *services* it offers (F_1, F_2). We refer to different functions in the component's implementation that together implement a service ($F_{1,1}$, $F_{1,2}$, $F_{1,3}$) as *service fragments*. They handle incoming communications, e.g., client requests and server responses. We use this architecture to decompose observations, downwards in the middle column, and compose models, upwards in the right column, to reconstruct the system behavior.

The *behavior* of a system is the order in which the various communications can occur, and thus the order in which the corresponding functions can be called. The CMI approach requires observing the system behavior during one or more executions (Preparation). The resulting runtime observations in the form of execution logs, consisting of events, are the input to our approach. Each *event* represents the start or end of a function call. An *observation* is thus a possible order in which the events of the system can occur, as observed during an execution.

Using knowledge of the system architecture, the observations are first decomposed into observations pertaining to individual components (Step 1). Assuming that the beginning and the end of each service fragment can be identified, we decompose observations further into observations of individual service fragments (Step 2). Then, we infer finite state automata models of service fragments from their observations (Step 3), assuming that offered services may be repeatedly requested, and executed from start to end. A service fragment model captures the behavior of a service fragment, the possible orders in which a handler function may communicate with other components. The service fragment models are combined to form component models (Step 4), where each component repeatedly executes its services, non-preemptively, one at a time. A component model captures the behavior of a service fragment, the possible orders in which the different handler functions may be called, and how these handler functions then may communicate with other components. These component models are put in parallel to form a model that captures the system behavior (Step 6).

The learned system model may exhibit behavior that the real system does not, e.g., due to missing dependencies between service fragments. The CMI approach provides for optional refinement (Step 5), whereby behavioral constraints derived from the software architecture and expert knowledge can be added to the component models, in a generic and structured way. Injecting such behavior allows turning stateless services into stateful ones, removing non-system behavior from the models.

The composition in Step 6 can be performed in various ways, depending on assumptions about the way the system is composed of components, i.e., either synchronously or asynchronously, with varying buffering and scheduling policies. In Figure 3.1, this is visualized as the composition of component models with explicit buffer models (B_1, B_2).

The approach is fully automated, except for optional step 5. Step 5 requires user-provided input, but is otherwise automated as well.

3.1.1 Running example

Figure 3.2 shows two examples of observations of the behavior of a component-based system. The horizontal axis represents passage of time, from left to right. The system consists of three components, C_1, C_2, and C_3, shown on the vertical axis. In Figure 3.2a, component C_1 receives an incoming request req_f from the environment, and in response executes its service fragment (function) f, illustrated by the horizontal bar, ultimately leading to a reply rep_f. During the execution of function f, component C_1 executes function g and component C_3 handles it. This involves a remote procedure call (arrow in the figure), with request req_g being sent to component C_3, and after C_3 has executed function g, its reply rep_g being received by C_1. C_1 similarly calls h on component C_2. After C_1 becomes idle again, a second request req_z is received, and handled in service fragment z, leading to a reply rep_z, this time without involving other components.

In the figure, the stacked bars for each component represent call stacks of nested function calls. The bottom bars represent service fragment function executions. Complete call stacks are visualized in the figure by enclosing them in blue dashed rectangles. From the start of a service fragment's call stack, until its end where it is idle again, this represents a single observation of its behavior.

Figure 3.2b shows a different observation of the system, where request req_f is handled asynchronously. Again, services from components C_3 and C_2 are requested, but now the system does not wait for their replies. Instead, it completes service fragment f and proceeds to handle request req_z. When the responses from the other components come in (rep_h and rep_g), it handles these in separate service fragments (hr and gr). Having received both responses, it sends reply rep_f as part of the last service fragment, by a call to function fr. Here service fragments f, hr and gr together implement a service of C_1, offered via req_f.

Such system behaviors can be observed in the form of an execution log, sequences of events in the order in which they occurred in the system. Each start and end of a function execution (bars in Figure 3.2) has an associated event. We identify an event by its executing component, the related function, and whether it represents the start ($\uparrow$) or the completion ($\downarrow$) of its execution. E.g., event $f^{\uparrow}_{C_1}$, abbreviated to $f^{\uparrow}_{1}$, denotes the start of function f on component C_1. For instance, for the synchronous function call of function g from component C_1, being handled

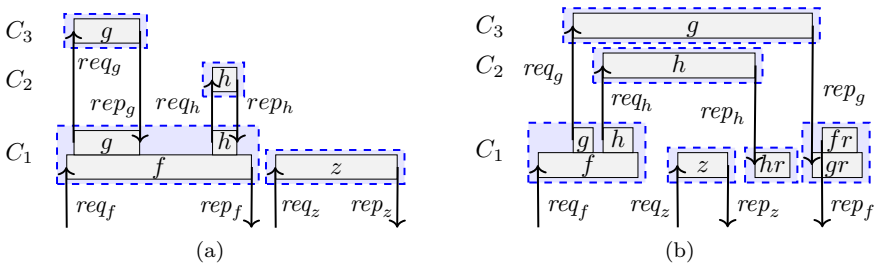
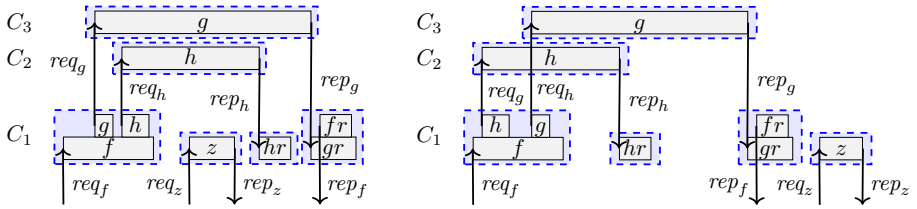


Figure 3.2: Observations showing client request req_f being handled by component C_1 through calls g and h to its servers, (a) synchronously and (b) asynchronously.



(a) w_1 (b) w_2

Figure 3.3: Observations for the running example.

by component C_3, as shown in Figure 3.2a, there are thus four events: $g_1^{\uparrow}$ and $g_1^{\downarrow}$ for the start and end of the function call on C_1, and $g_3^{\uparrow}$ and $g_3^{\downarrow}$ for the start and end of the handler function on C_3. The g call on C_1 through $g_1^{\uparrow}$ leads to the handler for g starting to execute on C_2, as represented by $g_3^{\uparrow}$. Once the execution of g finishes on C_2, represented by $g_3^{\downarrow}$, this leads to the call on C_1 ending as well, represented by $g_1^{\downarrow}$. Where appropriate, we identify service fragments by their start events, e.g., $f_1^{\uparrow}$, $z_1^{\uparrow}$, $hr_1^{\uparrow}$, $gr_1^{\uparrow}$, $h_2^{\uparrow}$, and $g_3^{\uparrow}$, for the service fragments of Figure 3.2b.

Our running example has two observations, w_1 and w_2. The first one, w_1, is the behavior from Figure 3.2b, which for convenience is repeated in Figure 3.3a. w_1 consists of the start and end events of each function execution (the grey bars in the figure), in the order that they occurred in time. That is:

$$w_1 = \langle f_1^{\uparrow}, g_1^{\uparrow}, g_3^{\uparrow}, g_1^{\downarrow}, h_1^{\uparrow}, h_2^{\uparrow}, h_1^{\downarrow}, f_1^{\downarrow}, z_1^{\uparrow}, z_1^{\downarrow}, h_2^{\downarrow}, hr_1^{\uparrow}, hr_1^{\downarrow}, g_3^{\downarrow}, gr_1^{\uparrow}, fr_1^{\uparrow}, fr_1^{\downarrow}, gr_1^{\downarrow} \rangle$$

The second one, w_2, is shown in Figure 3.3b. w_2 is a variation of w_1, where the calls to g and h are reversed, and z is handled last. That is:

$$w_2 = \langle f_1^{\uparrow}, h_1^{\uparrow}, h_2^{\uparrow}, h_1^{\downarrow}, g_1^{\uparrow}, g_3^{\uparrow}, g_1^{\downarrow}, f_1^{\downarrow}, h_2^{\downarrow}, hr_1^{\uparrow}, hr_1^{\downarrow}, g_3^{\downarrow}, gr_1^{\uparrow}, fr_1^{\uparrow}, fr_1^{\downarrow}, gr_1^{\downarrow}, z_1^{\uparrow}, z_1^{\downarrow} \rangle$$

3.2 Preliminary definitions

This section introduces basic definitions that we build upon to place our CMI approach in a framework based on finite state automata and regular languages [32, 65, 108].

3.2.1 Finite state automata

Let Σ be a finite set of *symbols* or *events*, called an *alphabet*. A word (or string) w over Σ is a finite concatenation of events from Σ. Given a word w, we denote its length as $|w|$, and its i^{th} event as w_i. If words u and v are such that $uv = w$, then uv is a concatenation, u is a prefix of w, and v is a suffix of w.

The Kleene star closure of Σ, Σ^*, is the set of all finite words over Σ, including the empty word denoted as ε. The Kleene plus closure of Σ is defined as $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$. A *language* over Σ is a subset of Σ^*. Given two languages K, L over the same alphabet, concatenation language KL is $\{uv \mid u \in K, v \in L\}$. The repetition of a language L is recursively defined to be $L^0 = \{\varepsilon\}$, $L^{i+1} = L^i L$. Similarly, we define the repetition of a word w: $w^0 = \varepsilon$, $w^{i+1} = w^i w$. The Kleene star and plus closures of L are defined as $L^* = \bigcup_{n=0}^{\infty} L^n$ and $L^+ = \bigcup_{n=1}^{\infty} L^n$, respectively.

We represent inferred models using state machines, namely as DFAs:

Definition 3.1 (DFA)**.** *A* deterministic finite automaton *(DFA) A is a 5-tuple $A = (Q, \Sigma, \delta, q_0, F)$, with Q a finite set of states, Σ a finite alphabet, $\delta : Q \times \Sigma \hookrightarrow Q$ the partial transition function, $q_0 \in Q$ the initial state and $F \subseteq Q$ a set of accepting states.*

The transition function is extended to words such that $\delta : Q \times \Sigma^* \hookrightarrow Q$, by inductively defining $\delta(q, \varepsilon) = q$ and $\delta(q, wa) = \delta(\delta(q, w), a)$, for $w \in \Sigma^*$, $a \in \Sigma$. DFA $A = (Q, \Sigma, \delta, q_0, F)$ accepts word w iff state $\delta(q_0, w) \in F$. If w is not accepted by A, it is rejected. Set $\mathcal{L}(A) = \{w \in \Sigma^* \mid A \text{ accepts } w\}$ is the language of A.

A DFA is *minimal* iff every state $q \in Q$ is reachable, i.e. there is a $w \in \Sigma^*$ such that $\delta(q, w)$ is defined, and every two states $p, q \in Q$ $(p \neq q)$ can be distinguished, i.e. there is a $w \in \Sigma^*$ such that $\delta(p, w) \in F$ and $\delta(q, w) \notin F$, or vice versa.

Given a finite set of words $W \subseteq \Sigma^*$, a *Prefix Tree Automaton* $PTA(W)$ is a tree-structured DFA with $\mathcal{L}(PTA(W)) = W$, where common prefixes of W share their states and transitions. As a PTA is tree-structured, it is also acyclic.

Given two DFAs A_1, A_2 we define operations on DFAs with notations that reflect the effect on their resulting languages, i.e., $A_1 \cap A_2$, $A_1 \cup A_2$, and $A_1 \setminus A_2$ result in DFAs with languages $\mathcal{L}(A_1) \cap \mathcal{L}(A_2)$, $\mathcal{L}(A_1) \cup \mathcal{L}(A_2)$, and $\mathcal{L}(A_1) \setminus \mathcal{L}(A_2)$, respectively. In addition, we define synchronous composition:

Definition 3.2 (Synchronous composition)**.** *Given two DFAs $A_1 = (Q_1, \Sigma_1, \delta_1, q_{0,1}, F_1)$ and $A_2 = (Q_2, \Sigma_2, \delta_2, q_{0,2}, F_2)$, their synchronous composition, denoted $A_1 \| A_2$, is the DFA:*

$$A = (Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \delta, (q_{0,1}, q_{0,2}), F_1 \times F_2),$$

with $\delta((q_1, q_2), a)$ defined as:

$$\begin{cases} (\delta_1(q_1, a), \delta_2(q_2, a)) & \text{if } \delta_1(q_1, a), \delta_2(q_2, a) \text{ are defined,} \\ (\delta_1(q_1, a), q_2) & \text{if } \delta_1(q_1, a) \text{ is defined, and } a \notin \Sigma_2, \\ (q_1, \delta_2(q_2, a)) & \text{if } \delta_2(q_2, a) \text{ is defined, and } a \notin \Sigma_1, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Two DFAs A_1 and A_2 are language equivalent, $A_1 \Leftrightarrow_L A_2$, iff $\mathcal{L}(A_1) = \mathcal{L}(A_2)$. Under language equivalence, each of the operators $\diamond \in \{\|, \cup, \cap\}$ is both commutative and associative, i.e., $A_1 \diamond A_2 \Leftrightarrow_L A_2 \diamond A_1$ and $(A_1 \diamond A_2) \diamond A_3 \Leftrightarrow_L A_1 \diamond (A_2 \diamond A_3)$.

To reason about components of a synchronous composition, we define word projection:

Definition 3.3 (Word projection)**.** *Given a word w over alphabet Σ, and a target alphabet Σ', we define the projection $\pi_{\Sigma'}(w) : \Sigma^* \to \Sigma'^*$ inductively as:*

$$\pi_{\Sigma'}(w) = \begin{cases} \varepsilon & \text{if } w = \varepsilon, \\ \pi_{\Sigma'}(v) & \text{if } w = va \text{ with } v \in \Sigma^*, a \notin \Sigma', \\ \pi_{\Sigma'}(v)a & \text{if } w = va \text{ with } v \in \Sigma^*, a \in \Sigma'. \end{cases}$$

This definition is lifted to sets of words: $\pi_{\Sigma'}(L) = \{\pi_{\Sigma'}(w) \mid w \in L\}$.

With word projection, we define *synchronization* of languages, which is commutative and associative:

Definition 3.4 (Synchronization)**.** *Given two languages $L_1 \subseteq \Sigma_1^*$ and $L_2 \subseteq \Sigma_2^*$, the* synchronization *of L_1 and L_2 is the language $L_1 \parallel L_2$ over $\Sigma = \Sigma_1 \cup \Sigma_2$ such that for all $w \in \Sigma^*$:*

$$w \in (L_1 \parallel L_2) \Leftrightarrow \pi_{\Sigma_1}(w) \in L_1 \wedge \pi_{\Sigma_2}(w) \in L_2.$$

From the definitions we derive:

Proposition 3.1. *Given two DFAs A_1 and A_2, their synchronous composition is homomorphic with the synchronization of their languages:*

$$\mathcal{L}(A_1 \parallel A_2) = \mathcal{L}(A_1) \parallel \mathcal{L}(A_2).$$

Corollary 3.2. *Given three DFAs A, A_1, and A_2, over alphabets Σ, Σ_1, and Σ_2, respectively, such that $A = A_1 \parallel A_2$ and $\Sigma = \Sigma_1 \cup \Sigma_2$, then for all $w \in \Sigma^*$:*

$$w \in \mathcal{L}(A) \Leftrightarrow \pi_{\Sigma_1}(w) \in \mathcal{L}(A_1) \wedge \pi_{\Sigma_2}(w) \in \mathcal{L}(A_2).$$

3.2.2 Formalizing concurrent behavior

To represent concurrent behavior, we briefly introduce Mazurkiewicz Trace theory [90]. Intuitively, events that occur in the alphabets of multiple automata synchronize in the synchronous composition (are dependent), while, e.g., internal non-communicating events and communications involving different components interleave (are independent), and can thus be reordered or *commuted*.

Formally, let *dependency* $D \subseteq \Sigma_D \times \Sigma_D$ be a symmetric reflexive relation over *dependency alphabet* Σ_D. Inversely, relation $I_D = (\Sigma_D \times \Sigma_D) \setminus D$ is the *independency* induced by D. The commutation of events is captured by binary relation $\sim_D$, with $u \sim_D v$ iff there are $x, y \in \Sigma_D^*$ and $(a, b) \in I_D$ such that $u = xaby$ and $v = xbay$. Mazurkiewicz *trace equivalence* for D, denoted as $\equiv_D$, is the reflexive transitive closure of $\sim_D$, i.e., $u \equiv_D v$ iff there exists a sequence $(w_0, \ldots, w_n)$ such that $w_0 = u$, $w_n = v$ and $w_i \sim_D w_{i+1}$ for $0 \leq i < n$. Mazurkiewicz *trace equivalence* for D can alternatively be defined as the least congruence $\equiv_D$ in the monoid Σ_D^* such that for all $a, b \in \Sigma_D$: $(a, b) \in I_D \Rightarrow ab \equiv_D ba$, i.e., the smallest equivalence relation that, in addition to the above, is preserved under concatenation: $u_1 \equiv_D u_2 \wedge v_1 \equiv_D v_2 \Rightarrow u_1 v_1 \equiv_D u_2 v_2$.

Equivalence classes over $\equiv_D$ are called *traces*. A trace $[w]_D$ for a word w is the set of words equivalent to w under D, i.e., all commutations of w under D. This definition is lifted to languages: $[\mathcal{L}]_D = \{[w]_D \mid w \in \mathcal{L}\}$. We drop subscript D if it is clear from the context. Language iteration is extended to traces by defining concatenation of $[u]_D, [v]_D \in [\Sigma_D^*]_D$ as $[u]_D[v]_D = [uv]_D$, with $[u]_D$ a prefix of $[uv]_D$, and $[v]_D$ a suffix of $[uv]_D$.

Given a set T of traces, $\operatorname{lin} T$ is the *linearization* of T, i.e., the set $\{w \in \Sigma_D^* \mid [w]_D \in T\}$. For any string language L, if $L = \operatorname{lin}[L]_D$ then L is *consistent* with D, as opposed to when $L \subset \operatorname{lin}[L]_D$. A consistent language thus has all the commutations allowed by D. If the language $\mathcal{L}(A)$ of automaton A is consistent with D, A has *trace language* $\mathcal{T}(A) = [\mathcal{L}(A)]_D$, i.e., $\mathcal{L}(A) = \operatorname{lin}[\mathcal{L}(A)]_D$.

As an example, consider $\Sigma_D = \{a, b, c\}$ and $D = \{a, b\}^2 \cup \{a, c\}^2 = \{(a, a), (a, b), (a, c), (b, a), (b, b), (c, a), (c, c)\}$. As b and c can then occur independently, $I_D = \{(b, c), (c, b)\}$. Word $abbca$ is part of trace $[abbca]_D = \{abbca, abcba, acbba\}$, which confirms that commuting b and c results in the same trace.

We rely on an additional result from Mazurkiewicz [90]:

Proposition 3.3. *Given dependency D and words $u, v \in \Sigma_D^*$, then for any alphabet Σ, we have: $u \equiv_D v \Rightarrow \pi_\Sigma(u) \equiv_D \pi_\Sigma(v)$.*

3.2.3 Asynchronous compositions

In addition to synchronous composition, we introduce asynchronous composition [6, 24]. This makes use of explicit buffers to pass messages from a sender to a receiver. For an asynchronous composition of DFAs $A_1, \ldots, A_n$, we assume component A_i, $1 \leq i \leq n$, has alphabet Σ_i, partitioned in sending event, receiving events, and internal events, $\Sigma_i^!$, $\Sigma_i^?$, and Σ_i^τ, respectively. Sending events are at the start of a communication, receiving events are at the end of a communication, and all other events are internal events. For instance, in Figure 3.3a, $g_1^\uparrow$ is a sending event, $g_3^\downarrow$ a receiving event, and $g_1^\downarrow$ an internal event. Each message has a unique sender and receiver, $\Sigma_i^! \cap \Sigma_j^! = \emptyset$ and $\Sigma_i^? \cap \Sigma_j^? = \emptyset$, for $i \neq j$. The receiver is assumed to exist, and to be different from the sender, $a \in \Sigma_i^! \Rightarrow \exists_{j \neq i} : a \in \Sigma_j^?$. Finally, we assume internal events are unique to a component, $\Sigma_i^\tau \cap \Sigma_j = \emptyset, i \neq j$. We denote an alphabet under these assumptions as $\Sigma_i^{!,?,\tau}$.

The components communicate via buffers. A buffer is denoted as B_i. It represents, e.g., a FIFO buffer or a bag buffer. FIFO buffers are modeled as a list of events over $\Sigma_i^?$, with ε the empty buffer, where messages are added to the tail of the list and consumed from the head of the list. We define the asynchronous composition using (for now unbounded) FIFO input buffers as follows:

Definition 3.5. *Consider n DFAs $A_1, \ldots, A_n$, with $A_i = (Q_i, \Sigma_i^{!,?,\tau}, \delta_i, q_{0,i}, F_i)$, $1 \leq i \leq n$. The asynchronous composition A of $A_1, \ldots, A_n$ using FIFO buffers $B_i, \ldots, B_n$, denoted $A = \|_{i=1}^n (A_i \,\|\, B_i)$, is given as the (typically infinite) DFA:*

$$A = (Q, \Sigma, \delta, (q_{0,1}, \varepsilon, \ldots, q_{0,n}, \varepsilon), F_1 \times \cdots \times F_n)$$

with $Q = Q_1 \times (\Sigma_1^?)^* \times \cdots \times Q_n \times (\Sigma_n^?)^*$, $\Sigma = \bigcup_i \{a! \,|\, a \in \Sigma_i^!\} \cup \bigcup_i \{a? \,|\, a \in \Sigma_i^?\} \cup \bigcup_i \Sigma_i^\tau$, and $\delta : Q \times \Sigma \hookrightarrow Q$ such that for $q = (q_1, b_1, \ldots, q_n, b_n) \in Q$ and $q' = (q_1', b_1', \ldots, q_n', b_n') \in Q$ we have:

(send) $\delta(q, a!) = q'$ *if* $\exists_{i,j}$: *(i)* $a \in \Sigma_i^! \cap \Sigma_j^?$, *(ii)* $\delta_i(q_i, a) = q_i'$, *(iii)* $b_j' = b_j a$, *(iv)* $\forall_{k \neq i}\ q_k' = q_k$, *(v)* $\forall_{k \neq j}\ b_k' = b_k$.

(receive) $\delta(q, a?) = q'$ *if* $\exists_i$: *(i)* $a \in \Sigma_i^?$, *(ii)* $\delta_i(q_i, a) = q_i'$, *(iii)* $b_i = a b_i'$, *(iv)* $\forall_{k \neq i}\ q_k' = q_k$, *(v)* $\forall_{k \neq i}\ b_k' = b_k$.

(internal) $\delta(q, a) = q'$ *if* $\exists_i$: *(i)* $a \in \Sigma_i^\tau$, *(ii)* $\delta_i(q_i, a) = q_i'$, *(iii)* $\forall_{k \neq i}\ q_k' = q_k$, *(iv)* $\forall_k\ b_k' = b_k$.

Bag buffers are defined as a multiset over $\Sigma_i^?$. To use bags instead of FIFOs, we change: ε to $\emptyset$ in the definition of A, list type to multiset type in the definition

of Q, send rule clause (iii) to $b'_j = b_j \cup \{a\}$, and receive rule clause (iii) to $a \in b_i \wedge b'_i = b_i - \{a\}$.

With unbounded buffers, asynchronous compositions can have infinite state spaces. When buffers are bounded, the buffer models can be represented by a (finite) DFA [97], and hence the composition as well. We bound buffer B_i to k places, denoted B_i^k, by adding requirement $|b_j| < k$ (such that $|b'_j| \leq k$) to the send rule.

We do not discuss the construction of a DFAs for B_i^k, as it follows from the definition above. Then, the synchronous composition of such constructed DFAs $A_i \parallel B_i^k$ is equivalent to asynchronous composition $A_i \parallel B_i^k$ as in Definition 3.5, if for synchronous composition we differentiate $a!$ and $a?$ to prevent synchronization of communications to and from buffers, respectively.

The question whether there is a buffer capacity bound such that every accepted word of the unbounded composition is also accepted by the bounded composition is called *boundedness* and is generally undecidable [48]. However, if the asynchronous composition is 'deadlock free', i.e., an accepting state can be reached from every reachable state [81], then it is decidable for a given k whether the asynchronous composition is bounded to k [48].

3.3 CMI for synchronous composition

Recall the CMI approach introduced in Section 3.1, and its overview in Figure 3.1. We assume system execution observations are available (*Preparation* step). They are decomposed following the system architecture (Steps 1 and 2). Models are then inferred at the most detailed level, for service fragments (Step 3). Again following the architecture, inferred models are composed to obtain models at various levels of abstraction (Steps 4 – 6). In the previous sections we introduced the definitions and results with which we can now describe our approach in detail.

In this section, we consider a system with a synchronous composition of components. We later lift this restriction in Section 3.4, where we discuss asynchronous compositions. Note that even though we for now only consider synchronous composition, components may still synchronously or asynchronously request services of other components, being blocked or not blocked, respectively, while waiting for the reply. Similarly, components may still synchronously or asynchronously handle requests, either replying immediately or replying at a later time after handling other requests in the mean time.

Formally, in this section, we assume the system under study is a DFA $A = (Q, \Sigma, \delta, q_0, F)$, Σ consists of observable events, A is synchronously composed of n component DFAs, $A = A_1 \parallel \ldots \parallel A_n$, and observations $W \subseteq \mathcal{L}(A)$ are available to infer an approximation A' of A, with $W \neq \emptyset$. We use subscripts for component and service fragment instances, e.g., C_1 for the first component of a system, and prime symbols for inferred instances, e.g., A' for the inferred model for A.

3.3.1 Step 1: System decomposition

Informal description: We assume a fixed and known deployment of services on components, such that we can project the observations onto each component.

Formalization: We assume component alphabets Σ_i are known a-priori. W is projected to $\pi_{\Sigma_i}(W)$ for each component, $1 \leq i \leq n$. Recall that we denote events in Σ as f_i^s, with f a function, C_i a component, and $s \in \{\uparrow, \downarrow\}$ denoting the start or completion of a function execution, respectively. For instance, $f_1^\uparrow$ denotes the start of function f on component C_1. Hence, alphabet Σ_i for component C_i contains the events with subscript i.

Example: Consider again the running example from Section 3.1, with $W = \{w_1, w_2\}$. By projection on component C_1, we get two projected words, i.e., $\pi_{\Sigma_1}(W) = \{\langle f_1^\uparrow, g_1^\uparrow, g_1^\downarrow, h_1^\uparrow, h_1^\downarrow, f_1^\downarrow, z_1^\uparrow, z_1^\downarrow, hr_1^\uparrow, hr_1^\downarrow, gr_1^\uparrow, fr_1^\uparrow, fr_1^\downarrow, gr_1^\downarrow\rangle, \langle f_1^\uparrow, h_1^\uparrow, h_1^\downarrow, g_1^\uparrow, g_1^\downarrow, f_1^\downarrow, hr_1^\uparrow, hr_1^\downarrow, gr_1^\uparrow, fr_1^\uparrow, fr_1^\downarrow, gr_1^\downarrow, z_1^\uparrow, z_1^\downarrow\rangle\}$. By projection on component C_2, we obtain twice the same projected word, i.e., $\pi_{\Sigma_2}(W) = \{\langle h_2^\uparrow, h_2^\downarrow\rangle\}$. By projection on component C_3, we also obtain twice the same projected word, i.e., $\pi_{\Sigma_3}(W) = \{\langle g_3^\uparrow, g_3^\downarrow\rangle\}$.

3.3.2 Step 2: Component decomposition

Informal description: We assume that:

- Components are sequential, e.g., corresponding to a single operating system thread.
- Client requests (and server responses) can only be handled once the component is idle, and prior requests are finished, i.e., service fragments are executed non-preemptively.
- Events that start or end a service fragment can be distinguished.
- Components do not call the functions of their own interface, such that within a service fragment there are only communications to other components.

These assumptions enable us to decompose component observations into service fragment observations.

Formalization: A *task* or *task word* captures a possible execution behavior of a service fragment:

Definition 3.6 (Task)**.** *Let Σ be a partitioned alphabet $\Sigma^{s,o,e} = \Sigma^s \cup \Sigma^o \cup \Sigma^e$, with service fragment execution start events Σ^s, its corresponding end events Σ^e, and other events Σ^o. Word $w \in \Sigma^*$ is a* task *on component C_i iff $w = f_i^\uparrow v f_i^\downarrow$ with $f_i^\uparrow \in \Sigma^s$, $v \in \Sigma^{o*}$, and $f_i^\downarrow \in \Sigma^e$.*

The start and end events do not occur within the tasks, which means that no task for a service fragment can be a prefix of another task for that same service fragment.

We also define *task sequence*, a sequence of tasks, and *task set*, the set of tasks comprising such a sequence:

Definition 3.7 (Task sequence/set)**.** *Given alphabet $\Sigma^{s,o,e}$, a word $w \in \Sigma^*$ is a* task sequence *iff $w = w_1 \dots w_n$ with each w_i, $1 \leq i \leq n$, a task. The set $T(w) = \{w_i \mid 1 \leq i \leq n\}$ is the* task set *corresponding to w. Similarly $T(W) = \bigcup_{w \in W} T(w)$ for $W \subseteq \Sigma^*$. Identifying a service fragment by its start event $f \in \Sigma^s$, its task set is $T_f(w) = \{v \mid v \in T(w) \wedge w_1 = f\}$. $T_f(W) \subseteq T(W)$ is similarly defined.*

For each component C_i, based on the given assumptions, component observations $\pi_{\Sigma_i}(W)$ are task sequences. From these task sequences, we obtain for each service fragment $f \in \Sigma^s$ its task set $T_f(\pi_{\Sigma_i}(W))$, containing the various observed alternative executions of f.

Example: For our running example, we get for service fragments $f_1^{\uparrow}$, $z_1^{\uparrow}$, $hr_1^{\uparrow}$, $gr_1^{\uparrow}$, $h_2^{\uparrow}$, and $g_3^{\uparrow}$, their respective task sets. That is:

- $T_{f_1^{\uparrow}}(\pi_{\Sigma_1}(W)) = \{\langle f_1^{\uparrow}, g_1^{\uparrow}, g_1^{\downarrow}, h_1^{\uparrow}, h_1^{\downarrow}, f_1^{\downarrow}\rangle, \langle f_1^{\uparrow}, h_1^{\uparrow}, h_1^{\downarrow}, g_1^{\uparrow}, g_1^{\downarrow}, f_1^{\downarrow}\rangle\}$,
- $T_{z_1^{\uparrow}}(\pi_{\Sigma_1}(W)) = \{\langle z_1^{\uparrow}, z_1^{\downarrow}\rangle\}$,
- $T_{hr_1^{\uparrow}}(\pi_{\Sigma_1}(W)) = \{\langle hr_1^{\uparrow}, hr_1^{\downarrow}\rangle\}$,
- $T_{gr_1^{\uparrow}}(\pi_{\Sigma_1}(W)) = \{\langle gr_1^{\uparrow}, fr_1^{\uparrow}, fr_1^{\downarrow}, gr_1^{\downarrow}\rangle\}$,
- $T_{h_2^{\uparrow}}(\pi_{\Sigma_2}(W)) = \{\langle h_2^{\uparrow}, h_2^{\downarrow}\rangle\}$, and
- $T_{g_3^{\uparrow}}(\pi_{\Sigma_3}(W)) = \{\langle g_3^{\uparrow}, g_3^{\downarrow}\rangle\}$.

3.3.3 Step 3: Service fragment model inference

Informal description: We infer a DFA per service fragment. Assuming services may be requested repeatedly, each DFA allows its service fragment to repeatedly be completely executed from start to end.

Formalization: For service fragment $f \in \Sigma^s$, $TDFA_f$ constructs a *Task DFA* (TDFA) from $T_f(\pi_{\Sigma_i}(W))$, i.e., $TDFA_f(W, \Sigma^{s,o,e}, f) = A'_f = (Q, \Sigma_i, \delta, q_0, \{q_0\})$. It has language $\mathcal{L}(A'_f) = T_f(\pi_{\Sigma_i}(W))^*$, allowing repeated uninterrupted executions of its observed set of tasks.

An efficient way to implement $TDFA_f$ is to build $PTA(T_f(\pi_{\Sigma_i}(W)))$ with root q_0 and language $T_f(\pi_{\Sigma_i}(W))$. Since no task for a service fragment is a prefix of another task for the same service fragment, all leafs of the PTA tree are accepting states, and there are no other accepting states. Then, optionally, minimize the PTA to reduce any redundancy. This merges all common postfixes, while preserving the PTA's language. Minimizing a PTA typically results in a DFA that is not a PTA. Finally, to allow the observed tasks to be repeatedly executed, merge all accepting states into initial state q_0, which is then the one and only accepting state. State q_0 exists, since we assume a non-empty set of input observations, and each task consists of at least the start and end events. The final result is a Task DFA, which is a DFA, but never a PTA.

Example: The inferred Task DFAs for the running example's service fragments are shown in Figure 3.4.

3.3.4 Step 4: Service fragment generalization

Informal description: We assume components can repeatedly handle requests of their services. We also assume (for now, but we revisit this assumption in Section 3.3.5) that components consist of service fragments, and service executions are mutually independent. That is, executing one service fragment does not impact which service fragments may subsequently be executed or the order in which

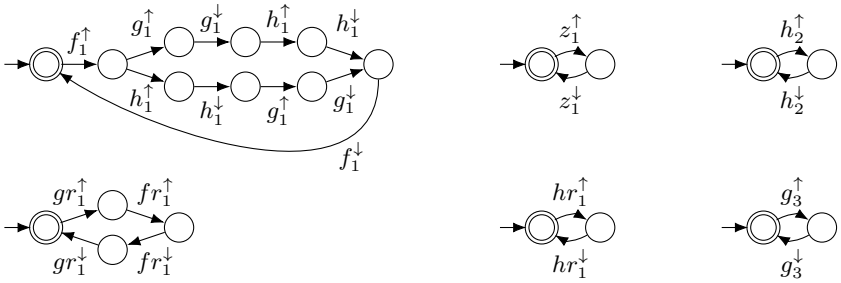


Figure 3.4: Task DFAs for the service fragments of the running example.

they may be executed, nor does it impose any restrictions on which of the possible behaviors of those service fragments may occur when they are executed.

We infer component models from service fragment models, generalizing the component behavior to repeated non-preemptive executions of the various service fragments. A component model can then execute any of the component's service fragments in arbitrary order, any number of times.

Formalization: For a component C_i, its service fragments are $\Sigma_i^s = \Sigma_i \cap \Sigma^s$. $TDFA_i$ constructs TDFA A_i' for component C_i, i.e., $TDFA_i(W, \Sigma^{s,o,e}, \Sigma_i^s) = A_i'$. It does so by merging the initial states of all TDFAs $A_f' = TDFA_f(W, \Sigma^{s,o,e}, f)$ for service fragments $f \in \Sigma_i^s$. Then $\mathcal{L}(A_i') = T(\pi_{\Sigma_i}(W))^*$. A_i' is deterministic, as all outgoing transitions from the initial states of TDFAs A_f', i.e., $f_i^{\uparrow}$, are unique. Given the assumptions of this step, we define such constructed TDFAs A_i' to be *stateless* component models.

Alternatively, such a stateless component model can be formed directly from component observations, rather than by partitioning the component observations into observations per service fragment, inferring service fragment models, and combining those service fragment models into a stateless component model. To do so, use $TDFA_f$ from Step 3, constructing the PTA from the component observations, rather than from service fragment observations.

Example: For the running example's components C_1, C_2, and C_3, the respective inferred Task DFAs A_1', A_2', and A_3', are shown in Figure 3.5. The component models for C_2 and C_3 are identical to their service fragment models, for $h_2^{\uparrow}$ and $g_3^{\uparrow}$, respectively, in Figure 3.4. For C_1, the initial states of its four service fragment models are merged.

3.3.5 Step 5: Stateful behavior injection

Informal description: In Step 4 we assumed stateless components with service fragments that are mutually independent. This assumption does not always hold in practice. Consider our running example (Figure 3.2b in Section 3.1.1). Service fragment f handles the responses for asynchronous calls g and h in service fragments gr and hr, respectively. Therefore, handling gr always comes after call g in f. This is ensured by the interaction with C_3, but it is not captured in the model for C_1.

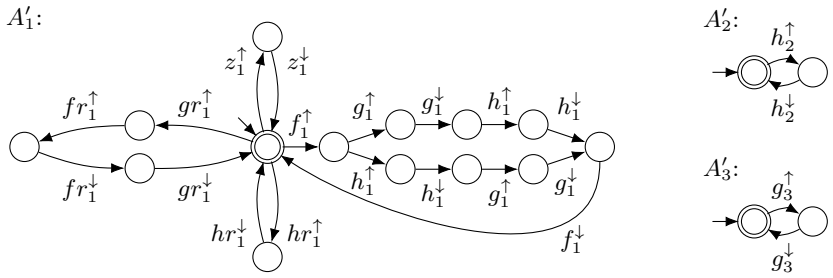


Figure 3.5: Task DFAs for the components of the running example.

Optional Step 5 allows to inject stateful behavior to obtain stateful component models that exclude behavior that cannot occur in the real system. As many varieties of component-based systems exist, our structured method for Step 5 allows to improve the inferred models based on injection of domain knowledge. This allows customization to fit a certain architecture or target system, and is not specific to our application of the CMI approach to a case study in Section 3.5. Our general domain knowledge injection method for instance allows one to capture the general property that 'a response must follow a request' (like *gr* after *g* in the running example).

Figure 3.6 visualizes the approach. We compose TDFAs inferred in Steps 1 – 4 with additional, typically small, automata, which specify explicitly which behavior we add, constrain or remove. Multiple different composition operators are supported: union, intersection, synchronous composition and (symmetrical) difference (see Section 3.2). The injected automata are either manually specified, or obtained by a miner [22, 107], an automated procedure that uses the observations as input.

Additional properties that should be added often apply in identical patterns across the whole system. To allow modeling them only once, DFAs with parameters are used, e.g., for the pattern of a request and reply. The parameterized DFA (template) is instantiated with specific events, e.g., user-provided request/reply pairs, to obtain the DFAs to inject.

Formalization: We define substitution, which renames parameter events to provided events:

Definition 3.8 (Substitution)**.** *Given a parameterized DFA* $A = (Q, \Sigma, \delta, q_0, F)$*, and events* $p \in \Sigma$*,* $a \notin \Sigma$*, the substitution of* a *for* p *in* A*, denoted* $A[p := a]$*, is*

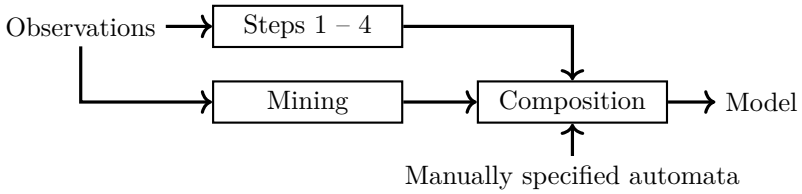


Figure 3.6: Generic method to inject specific domain knowledge in Step 5.

(a) P_1 (request before reply) (b) P_2 (server replies before client reply)

Figure 3.7: Parameterized property automata, as examples for Step 5.

defined as $A[p := a] = (Q, (\Sigma \setminus \{p\}) \cup \{a\}, \delta[p := a], q_0, F)$*, with* $\delta[p := a](q, c) = \delta(q, p)$ *if* $c = a$*, and* $\delta(q, c)$ *otherwise.*

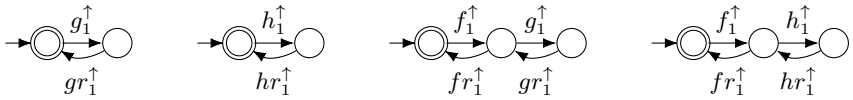
In general the order of substitutions matters. We apply them in the given order, and only replace parameters by concrete events, assuming both sets are disjoint.

Example pattern 1 (request before reply): For asynchronous calls, a reply must follow a request, e.g., *gr* after *g*. We model this property as DFA P_1 in Figure 3.7a. Parameter *Req* represents a request, *Reply* a reply. To enforce the property, we compose the inferred TDFAs A'_i with P_1, for every request and its corresponding reply in the system: $A''_i = A'_i \parallel (\|_{a \in reqs} P_1[Req := a][Reply := R(a)])$, where *reqs* is the set of requests, and $R : \Sigma_i \to \Sigma_i$ maps requests to their replies. By Corollary 3.2, for any $a \in reqs$: $\pi_{\{a,R(a)\}}(\mathcal{L}(A'')) = \{(a.R(a))^n \mid n \in \mathbb{N}\}$. This correctly models the informally given property, assuming at most one outstanding request for each a. For our running example, we get: $reqs = \{g_1^\uparrow, h_1^\uparrow\}$, $R(g_1^\uparrow) = gr_1^\uparrow$, and $R(h_1^\uparrow) = hr_1^\uparrow$. The instantiated DFAs $P_1^g = P_1[Req := g_1^\uparrow][Reply := gr_1^\uparrow]$ and $P_1^h = P_1[Req := h_1^\uparrow][Reply := hr_1^\uparrow]$ are shown in Figures 3.8a and 3.8b, respectively.

Example pattern 2 (server replies before client reply): For a service execution, often nested requests should have been replied before finishing the service. We model this property as DFA P_2 in Figure 3.7b. Upon receiving a client request *cReq*, P_2 goes to state 2. If during this service, a request *sReq* is sent to a server, it must be met with reply *sReply* to get out of state 3 and allow the original service to reply to its client with *cReply*. For our running example, we get:

- $P_2^g = P_2[cReq := f_1^\uparrow][cReply := fr_1^\uparrow][sReq := g_1^\uparrow][sReply := gr_1^\uparrow])$, and
- $P_2^h = P_2[cReq := f_1^\uparrow][cReply := fr_1^\uparrow][sReq := h_1^\uparrow][sReply := hr_1^\uparrow])$.

The instantiated DFAs P_2^g and P_2^h are shown in Figures 3.8c and 3.8d, respectively.



(a) P_1^g (b) P_1^h (c) P_2^g (d) P_2^h

Figure 3.8: Automata to inject in Step 5, for the running example.

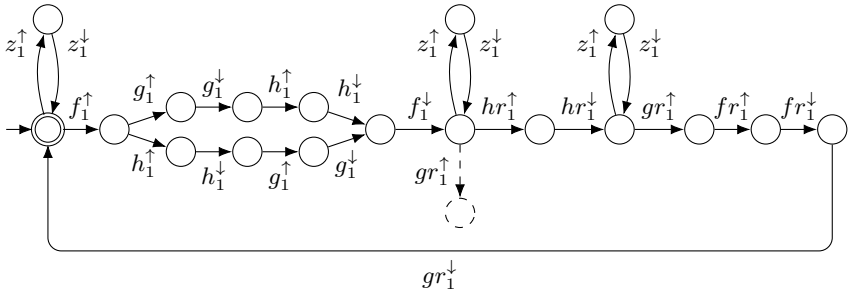


Figure 3.9: DFA for component C_1 of the running example, after injecting domain knowledge: A_1'' (solid and dashed parts) and A_1''' (solid parts only).

Running example: For the running example, we have four automata to inject for C_1, and none for C_2 and C_3. A_2' and A_3' thus remain as they are in Figure 3.5. For C_1, we compute $A_1'' = A_1' \parallel P_1^g \parallel P_1^h \parallel P_2^g \parallel P_2^h$. From A_1'', we prune all states from which no accepting state can be reached, to obtain A_1'''. The result is shown in Figure 3.9, where the parts of the model that are pruned away are indicated by dashed lines.

The examples show that domain knowledge can be added explicitly, straightforwardly, and with parameterized DFAs and miners also scalably.

3.3.6 Step 6: Component composition

Informal description: The last step is to form system models by composing the obtained stateless and/or stateful component models (from Steps 4 and 5). For now, we use a synchronous composition. We later consider an asynchronous composition in Section 3.4.

Formalization: So far we have used unique events per component, such as, e.g., $f_1^\uparrow$. Properly capturing component synchronization requires that we ensure the correct communications when one component uses the services of another component. For our running example (see Figure 3.3), we have a communication (arrow) from $g_1^\uparrow$ to $g_3^\uparrow$. To ensure correct synchronization in the synchronous composition (see Definition 3.2), we use the same event for both of them. For instance, we combine: $g_1^\uparrow$ and $g_3^\uparrow$ to the synchronization event '$g_1^\uparrow \triangleright g_3^\uparrow$', representing the start of call g on C_1 leading to the immediate start of a handler for g on C_3. The synchronous composition can then be applied directly, to obtain $A' = A_1' \parallel A_2' \parallel \ldots \parallel A_n'$, as per Definition 3.2.

Example: For the running example, we perform the event renaming on A_1''', A_2', and A_3', to obtain A_1'''', A_2'', and A_3'', respectively. We then compute composition $A' = A_1'''' \parallel A_2'' \parallel A_3''$. The resulting system model A' is shown in Figure 3.10. In this case, the result is identical to the solid part of the model in Figure 3.9, except for the renaming. Since components C_2 and C_3 have call stacks with only one bar, their behavior is already captured by C_1. Hence, for this particular running example, merging A_2'' and A_3'' with A_1''' results in A_1'''.

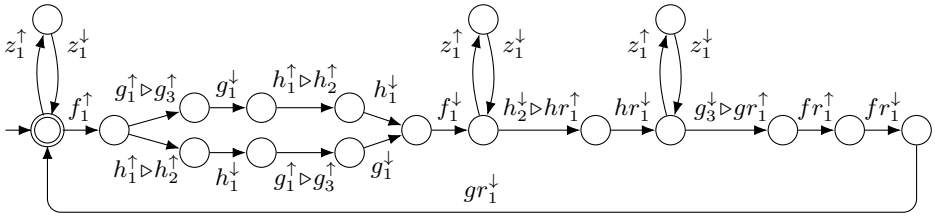


Figure 3.10: The inferred system model for the running example, DFA A'.

3.3.7 Analysis of the CMI approach

We analyze the CMI approach using synchronous composition as described in this section. We consider Steps 1 – 4 and 6, to infer service fragment, component, and system models. Step 5 is left out, as its effects on the behavior of the inferred models depend greatly on the domain knowledge being injected, which is manually specified by users or is mined by a user-defined miner. We thus forego injecting stateful behavior, and consider only stateless systems. That is, we consider systems that consist of stateless components, which allow their service fragments to be repeatedly executed in any order, without further restrictions. And those components are composed by putting them in parallel, without any additional restrictions imposed on their composition. Furthermore, the system must adhere to the various assumptions listed so far in Section 3.3.

We analyze four properties. Firstly, the inferred models at least accept the input observations. Secondly, they generalize beyond those observations. But thirdly, they do not over-generalize beyond the system behavior. Fourthly, adding additional observations only increases their behavior. Proving these properties is future work, but some proofs can be found in [62].

1) Accept at least the input observations

For a stateless system whose behavior is represented by DFA A, input observations $W \subseteq \mathcal{L}(A)$ are behaviors that A exhibits. A system model A' that is inferred from these observations, should at the very least accept them:

Proposition 3.4. *Consider a stateless system model A and observations $W \subseteq \mathcal{L}(A)$. System model A', which is inferred from those observations, then accepts those observations, i.e., $W \subseteq \mathcal{L}(A')$.*

Similarly, inferred component models should accept the component observations, and inferred service fragment models should accept the service fragment observations.

2) Generalize beyond the input observations

The inferred models should not only accept the input observations, but should also generalize beyond them. The inferred service fragment, component, and system models each generalize based on certain assumptions.

By assuming that services may be requested repeatedly, an inferred service fragment model allows repeated non-preemptive executions of the service fragment. That is, it allows any of its observed tasks to be executed in arbitrary order, any number of times:

Proposition 3.5. *Consider service fragment observations $T_f(\pi_{\Sigma_i}(W))$. Service fragment model A'_f, which is inferred from those observations, then allows tasks $T_f(\pi_{\Sigma_i}(W))$ to be repeatedly executed, i.e., $\mathcal{L}(A'_f) = T_f(\pi_{\Sigma_i}(W))^*$.*

By assuming components can repeatedly handle requests of their services, and that components are stateless and therefore service fragments are mutually independent (thus ignoring Step 5), an inferred component model allows repeated non-preemptive executions of its various service fragments. That is, it allows any of its service fragments to be executed in arbitrary order, any number of times:

Proposition 3.6. *Consider component observations $\pi_{\Sigma_i}(W)$. Component model A'_i, which is inferred from those observations, then allows tasks $T(\pi_{\Sigma_i}(W))$ to be repeatedly executed, i.e., $\mathcal{L}(A'_i) = T(\pi_{\Sigma_i}(W))^*$.*

By assuming a synchronous composition of components, an inferred system model allows independent events to commute. The concurrency between (stateless) components is used to accept all valid alternative interleavings that were not directly observed. Inferred system models at least generalize to traces using Mazurkiewicz trace theory:

Proposition 3.7. *Consider system observations W. Stateless system model $A' = A'_1 \parallel \ldots \parallel A'_n$, which is inferred from those observations, and constructed by synchronously composing inferred component models A'_i, consistently generalizes to traces, i.e., $\mathcal{L}(A') = \lim \mathcal{T}(A') \supseteq \lim T([W]_D)^*$.*

This proposition depends on dependency D, which we define such that it matches the synchronous parallel composition (see Definition 3.4): $D = \bigcup_{i=1}^{n} \Sigma_i^2$. That is, for every pair of events $e_1, e_2 \in \Sigma_i$, there is a dependency, i.e., $(e_1, e_2) \in \Sigma_i^2 = \Sigma_i \times \Sigma_i$. And the union of all these dependencies of the various components is the dependency set D.

For parallel composition, each component model restricts the system model to its allowed behavior, preserving the sequential nature of the components. Therefore, all events *within* a component are made dependent in D, preventing them from commuting. Events are also dependent on themselves, to ensure that D is reflexive.

Shared events for communications *between* components synchronize to a single event in the parallel composition, preserving communication behavior. Since in the composed system this concerns a single event, it does not impact the definition of D, which is only concerned with the dependencies between events.

The parallel composition allows non-shared events of components to interleave. Therefore, all events of a component that are not shared with any of the other components are not dependent in D. They are thus independent, and may commute, allowing all valid interleavings between components.

Proposition 3.7 uses an extended definition of T for *task sequence traces*:

Definition 3.9 (Task sequence trace)**.** *Consider a dependency* $D = \bigcup_{i=1}^{n} \Sigma_i^2$ *and alphabet* $\Sigma^{s,o,e}$*. A trace* $[t] \in [\Sigma_D^*]$ *is a* task sequence trace *iff* $\forall_{i=1}^{n}\ \pi_{\Sigma_i}(t)$ *is a task sequence for* $\Sigma^{s,o,e}$*. Its task set* $T([t])$*, is given as* $\{[t_j] \mid [t_1 \dots t_m] = [t]$ *and* t_j *a task for* $1 \leq j \leq m\}$*.*

Characterizing the full generalization remains an open problem.

3) No over-generalization

The inferred models should not over-generalize, i.e., should not allow more behavior than the actual system. We found that this is an essential property for industrial engineers to place their trust in the inferred models. Typically, neither heuristic-based approaches that infer models from logs, nor active automata learning approaches, satisfy this property. The CMI approach is specifically designed to avoid over-generalization, by only allowing generalization when it is known not to lead to over-generalization, based on knowledge of the system.

That is, with the assumptions we place on the architecture and deployment of the systems that we consider, we can formulate the following theorem, which is a main result of our approach:

Theorem 3.8. *Consider a stateless system model* A *and observations* $W \subseteq \mathcal{L}(A)$*. System model* A'*, which is inferred from those observations, then does not over-generalize, i.e.,* $\mathcal{L}(A') \subseteq \mathcal{L}(A)$*.*

Similarly, inferred component and service fragment models should not over-generalize.

If complete observations $W = \mathcal{L}(A)$ are used as input, then the approach should infer complete models, i.e., $\mathcal{L}(A') = \mathcal{L}(A)$. Providing complete observations is infeasible, as $\mathcal{L}(A)$ is generally not a finite set. However, the CMI approach generalizes beyond the input observations. Providing complete observations is therefore not needed, and it could already be sufficient to observe all unique service fragment behaviors. It remains an open problem to characterize the minimal observations needed as input to ensure complete models are inferred.

4) Robust under additional observations

The approach should be robust under additional observations, i.e., model inference from the same observations or more observations, leads to inferred models with the same or more behavior, so never less behavior:

Proposition 3.9. *Consider a stateless system model* A*, observations* $W \subseteq \mathcal{L}(A)$*, and system model* A' *inferred from* W*. Also consider observations* $V \subseteq \mathcal{L}(A)$*, with* $W \subseteq V$*, and system model* A'' *inferred from* V*. Then* $\mathcal{L}(A') \subseteq \mathcal{L}(A'')$*.*

Similarly, inferred component models should be robust under additional component observations, and inferred service fragment models should be robust under additional service fragment observations.

3.4 CMI for asynchronous composition

In Section 3.3 we considered systems consisting of synchronously composed components. This section considers the CMI approach applied to systems with asynchronously composed components.

We now assume the system is a DFA A, asynchronously composed of components A_i, using buffers B_i bounded to capacity b_i. Per Section 3.2.3 we model this as a synchronous composition with buffer automata, $A = \|_{i=1}^{n}(A_i \,\|\, B_i^{b_i})$. We shorten $B_i^{b_i}$ to B_i, leaving the buffer size implicit, in case it is not relevant. Then A_i has alphabet $\Sigma_i^{!,?,\tau}$, $\Sigma_{B_i} = \Sigma_{B_i}^{?} \cup \Sigma_{B_i}^{!}$, $\Sigma_{B_i}^{?} = \{a! \mid a? \in \Sigma_i^{?}\}$, and $\Sigma_{B_i}^{!} = \Sigma_i^{?}$. As an example, Figure 3.11 shows the events used to communicate between two components, via a buffer, for event $h_1^{\uparrow} \triangleright h_2^{\uparrow}$. For this example, $\Sigma_1^{!} = \Sigma_{B_2}^{?} = \{h_1^{\uparrow} \triangleright h_2^{\uparrow}!\}$ and $\Sigma_{B_2}^{!} = \Sigma_2^{?} = \{h_1^{\uparrow} \triangleright h_2^{\uparrow}?\}$. We furthermore assume system observations $W \subseteq \mathcal{L}(A)$ as before, still with $W \neq \emptyset$.

To infer A', we first infer component models A_i', reusing Steps 1 – 5 from the synchronous case of Section 3.3. Then, we model buffer DFAs $B_i'^{b_i}$ as in Section 3.2.3, according to the buffer type (FIFO or bag) and capacity bound b_i. A capacity lower bound $b_{i,w}$ for each buffer B_i is inferred from W, by considering the maximally occupied buffer space along each observed word $w \in W$. That is, $b_{i,w} = \max_{1 \leq i \leq |w|}(|\pi_{\Sigma_{B_i}^{?}}(w_1 \dots w_i)| - |\pi_{\Sigma_{B_i}^{!}}(w_1 \dots w_i)|)$. Then, the overall lower bound b_i for B_i is inferred, as $b_i = \max_{w \in W} b_{i,w}$. Finally, A' is given as synchronous composition $A' = \|_{i=1}^{n}(A_i' \,\|\, B_i'^{b_i})$.

Just as for the synchronous case, we analyze for this asynchronous version of Step 6, that A' accepts W, does not over-generalize, and is robust under additional observations:

Proposition 3.10. *Consider DFA $A = \|_{i=1}^{n}(A_i \,\|\, B_i)$ with each A_i stateless, B_i a FIFO (or bag) buffer, $W \subseteq \mathcal{L}(A)$, DFA $A' = \|_{i=1}^{n}(A_i' \,\|\, B_i'^{b_i})$ with $A_i' = TDFA_i(W, \Sigma^{s,o,e}, \Sigma_i^{s})$, and $B_i'^{b_i}$ a FIFO (or bag) buffer with $b_i = \max_{w \in W} b_{i,w}$. Then $W \subseteq \mathcal{L}(A') \subseteq \mathcal{L}(A)$. For DFA A'', similarly composed and obtained from observations V, with $W \subseteq V$, then holds $\mathcal{L}(A') \subseteq \mathcal{L}(A'')$.*

With this approach we need to know a-priori the kind of buffers used in our system. To understand whether our choices were correct we experimented with various buffer types. For instance, using a single FIFO buffer between each pair of components can lead to an issue shown in Figure 3.12. If req_{C_2} is sent before rep_g, the FIFO order enforces reception of req_{C_2} before rep_g, while the TDFA for C_3 has to handle rep_g before starting a new service with req_{C_2}, leading to a deadlock. In order to resolve this problem, and other mismatches with ASML's middleware, we use a FIFO buffer per client that a component communicates with, and a bag buffer per server [62].

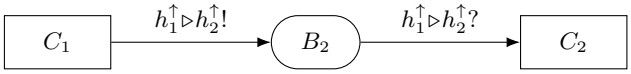


Figure 3.11: Asynchronous communication for event $h_1^{\uparrow} \triangleright h_2^{\uparrow}$ from component C_1 to component C_2, via buffer B_2.

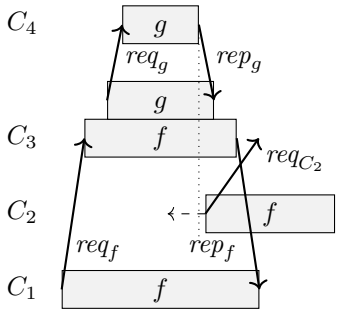


Figure 3.12: Example of limited commutations for FIFO buffers.

3.5 CMI in practice

We demonstrate our CMI approach by applying it to a case study at ASML. ASML designs and builds machines for lithography, which is an essential step in the manufacturing of computer chips.

3.5.1 System characteristics

Our CMI approach requires system observations as input. The middleware in ASML's systems has been instrumented, allowing us to extract Timed Message Sequence Charts (TMSCs) from executions. A TMSC [73] is a formal model for system observations, akin to what we described in Section 3.1. The TMSC formalism implies that the system can be viewed as a composition of sequential components bearing nested, fully observed, non-preemptive function executions. We can therefore obtain observations W, component alphabets Σ_i, and partitioned alphabet $\Sigma^{s,o,e}$, from TMSCs, to serve as inputs to our approach.

For this case study, we infer a model of the exposure subsystem, which exposes each field (die) on a wafer. By executing a system acceptance test, and observing its behavior during the exposure of a single wafer, which spans about 11 seconds, a TMSC is obtained that consists of around 100,000 events for 33 components.

3.5.2 Model inference steps 1 – 4

We apply Steps 1 – 4 for our case study. Figure 3.13 shows for each component the number of events in the observations and the number of states of the inferred component model. For most components the inferred model is two orders of magnitude more compact than its observations, showing repetitive service fragment executions in these components.

Component 1 orchestrates the wafer exposure. Its model has the same size as its observation, as it spans a single task with over 5,000 events. For components 26 – 33, the small reduction is due to their limited observations.

3.5.3 Model inference step 5

We analyze the inferred models, assessing whether their behavior is in accordance with what we expect from the actual software and if not, what knowledge must

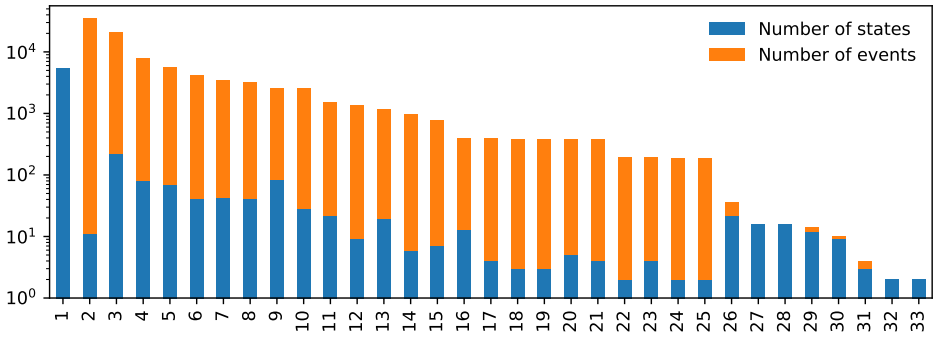


Figure 3.13: Per component, the number of states in its component model after Step 4, compared to the number of events in its observations.

be injected (according to Step 5).

Although the system matches an asynchronous composition of components as discussed in Section 3.4, we first apply our CMI approach for synchronous systems (Step 6 from Section 3.3). A synchronous composition has more limited behavior compared to an asynchronous composition, and hence a smaller state space. The resulting model can therefore be analyzed for issues more easily, while many such issues apply to both the synchronous and the asynchronous compositions.

We observe that component model 1, A'_1, has over 5,000 states, in a single task w. To reduce the model size, we analyze w for repetitions and reduce it [98]. We look for short $x, y, z \in \Sigma_1^*$ such that $w = xy^n z$ for some n. Then, we create reduced model A''_1, with $\mathcal{L}(A''_1) = (xyz)^*$. This reduces the model size by $|y| * (n-1)$ states to about 600 states for Component 1, in this case without limiting its synchronization with the other components.

As a second reduction, we remove DFA transitions that do not communicate and originate from a state which has a single outgoing transition, i.e., do not allow for a choice in the process. This is akin to process algebra axiom $a.\tau.b = a.b$, where τ is a non-synchronizing event [96]. Examples in Figure 3.14 are $g_7^{\downarrow}$ and $f_7^{\downarrow}$. This further reduces A''_1 from about 600 to about 300 states. Other components

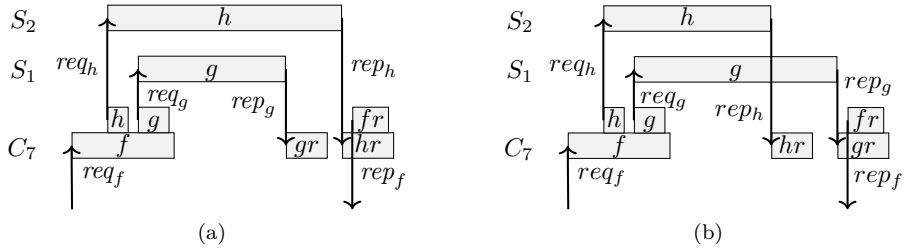


Figure 3.14: Component C_7 requires both rep_g and rep_h to reply to its client. (a) gr before hr, with fr in hr, and (b) hr before gr, with fr in gr.

are reduced as well.

With these two reductions, exploring the resulting state space becomes feasible, being approximately 10^{10} states. We look for deadlocks: states without outgoing transitions. As we use execution logs representing successful executions of the system, we expect no deadlocks in the inferred models. All deadlocks that we found where due to a synchronizing event, the counterpart of which could not be reached.

A first issue is illustrated in Figure 3.14. Component C_7 requests services g and h concurrently. Both are called asynchronously and can return in either order, with only the last reply leading to rep_f. The inferred stateless model does not capture the dependency between replies rep_g, rep_h, and rep_f. When in the learned model both, or neither, incoming replies are followed by rep_f, the system deadlocks, as the calling environment expects exactly one reply. This is solved by enforcing nested services to be finished before finishing f, as in Example 2 from Step 5 of Section 3.3.5.

A second issue is illustrated in Figure 3.15, where component C_{14} deals with server S_1^*. However, we do not observe the behavior of S_1^* directly, but merely through the messages observed at C_{14}. Server S_2^*, which is used by S_1^*, is not observed at all. The observation is shown in Figure 3.15a, and a possible perspective of the actual system behavior is shown in Figure 3.15b.

Since we do not observe the behaviors of S_1^* and S_2^*, the model misses the dependency between functions g and e_2. Now, e_2 has no incoming message and it is able to start 'spontaneously'. The actual system relies on the dependency, but it is missing in the learned behavior, leading to deadlocks. Knowing the missing dependency from domain knowledge, we inject it through Step 5 as a one-place buffer similar to Figure 3.7a, extended to multiple places as needed.

After resolving these issues, we analyze choices. From other observations we know that g is optional when performing f in component C_7 of Figure 3.14a. The inferred component model for C_7 thus contains tasks which have a common prefix, i.e., $\langle f_7^{\uparrow}, h_7^{\uparrow}, h_7^{\downarrow}, f_7^{\downarrow}\rangle$ (g is skipped), and $\langle f_7^{\uparrow}, h_7^{\uparrow}, h_7^{\downarrow}, g_7^{\uparrow}, g_7^{\downarrow}, f_7^{\downarrow}\rangle$ (g is called). After $h_7^{\downarrow}$ a choice arises, to either call g by $g_7^{\uparrow}$ or finish f by $f_7^{\downarrow}$. Such choices can enlarge the state space, and may not apply in all situations. We therefore

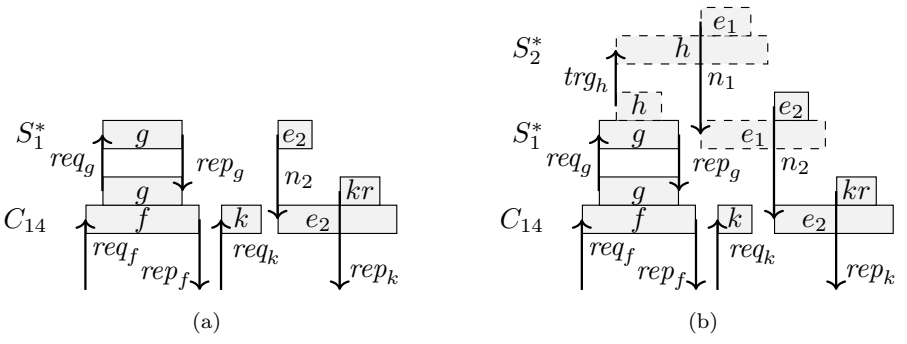


Figure 3.15: Component C_{14} communicates with components outside the observations, causing missing dependencies.

asked domain experts on which information in the software such a choice could depend. Together, we concluded that g is only skipped once, and this relates to the repetitions we observed for C_1, as g is skipped for one particular such iteration. We ensured the correct choice is made, by constraining the two options to their corresponding iterations on C_1, thus removing some further non-system behavior from the inferred model.

3.5.4 Model inference step 6

To the result of Step 5, we apply asynchronous composition (Step 6, Section 3.4). Using FIFO and bag buffers as described in that section, service fragments are allowed to commute due to execution and communication time variations.

For our case study, all inferred buffer capacities are either one or two, except component C_2, which has buffers with capacities up to 24. The low number of buffer places for most components, is due to the extensive synchronization on replies. C_2, the log component, only receives 'fire-and-forget'-type logging notifications without replies.

We verified that the resulting, improved, asynchronously composed model indeed accepts its input TMSC, i.e., the inferred model accepts the input observation from which it was inferred. This gives trust that the practically inferred model is in line with that observation. The inferred multi-level models were confirmed by domain experts as remarkably accurate, allowing them to discuss and analyze their system's behavior. This in stark contrast to models that were previously inferred using process mining and heuristics-based state machine learning, where they questioned the accuracy of the models instead of using them.

3.6 Conclusions and future work

We introduced our novel approach, Constructive Model Inference (CMI), which uses execution logs as input. Relying on knowledge of the system architecture, it allows learning the behavior of large concurrent component-based systems. The trace-theoretical framework provides a solid foundation. The ASML engineers that we worked with consider the state machine models resulting from our approach accurate, and the service fragment models in particular, to be highly intuitive. They see many potential applications, and are already using the inferred models to gain insight into their software behavior, as well as for change impact analysis (see Chapters 4 and 5).

Future work includes among others extending the CMI approach to inferring Extended Finite Automata and Timed Automata (see Section 7.2.1), further industrial application of the approach (see Chapters 4 and 5), and automatically deriving interface models from component models.

Chapter 4

Multi-level behavioral comparison methodology

Software-intensive systems, e.g., cyber-physical systems, become more and more complex. They often employ a component-based software architecture to manage their complexity. Over the years such systems continuously evolve by adding new features and addressing defects, more and more layers are built on top of each other [77], and components that are not well-maintained become legacy [85, 111].

Changing the software is often considered risky as any change can potentially break a system. If a software change leads to a system defect, then the impact can be tremendous due to system downtime and productivity loss [111]. This may even lead to software engineers becoming afraid to make changes for which they cannot properly foresee the impact on (other parts of) the system.

To reduce the risks, it is essential to understand the impact of software changes. However, for large complex industrial code bases consisting of tens of millions of lines of code, no single person has the complete overview. This makes it difficult to understand the impact of software changes on the overall system functionality [54]. This is especially true when the system can behave differently for different configurations and usage scenarios [132].

It is thus important that: 1) software developers understand how the system currently behaves for different configurations and usage scenarios, and 2) they understand how software changes impact that system behavior.

To address these needs, in this chapter we introduce a novel multi-level methodology for behavioral comparison of (large) software-intensive systems. The power of our methodology is that it quickly guides users to (the parts of the system with) relevant differences. This avoids the laborious and error-prone practice of looking into many thousands of lines of code, or plough through gigabytes of execution logs. Our method is fully automated, making it possible to consider huge (sub-)systems, for which due to their sheer size it is practically impossible to compare their behavior manually.

Our methodology is based on comparing state machine models rather than source code or execution logs, which makes it generally applicable. State machines can compactly and intuitively represent system behavior as a collection of software

function calls and the order in which they are called. Such models are general and can be obtained by any means of model learning or construction (see also Chapters 2 and 3).

Methods to compare state machines can be divided into two classes that complement each other [126]. Language-based methods compare state machines in terms of their allowed sequences of function calls, while structure-based methods compare them in terms of their states and transitions.

However, two important things are missing in the literature: 1) a single automated method integrating these individual methods to allow large-scale industrial application, and 2) an approach to inspect the resulting differences at various levels of detail, and step by step zoom in on relevant differences, to manage the complexity of huge systems. Our methodology tackles both these challenges.

Our methodology takes any number of sets of state machines representing software behavior of, e.g., different software versions, different configurations or different usage scenarios. We automatically compare the provided sets by comparing the languages and structures of their state-machine models. The comparison results can be inspected at six levels of abstraction, ranging from very high-level differences to very detailed ones. Users are guided through the differences in a step by step fashion tailored to allow them to zoom in on relevant behavioral differences, wasting no time on irrelevant ones.

We empirically evaluate the practical potential of our methodology through a qualitative exploratory field study [110, 118]. Using multiple case studies at ASML, a leading company in developing lithography systems, we demonstrate that our approach can be applied to large industrial (sub-)systems, provides developers and architects insight into their behavioral differences, and allows them to find unintended regressions.

The remainder of this chapter is organized as follows. In Section 4.1 we introduce the concepts, definitions and methods on which we build our methodology. Section 4.2 introduces our methodology, both conceptually and formally. We evaluate our methodology in Section 4.3, before concluding in Section 4.4.

4.1 Background

4.1.1 Software behavior

Programming languages typically have a notion of *function*, *procedure* or *method*. The behavior of software implemented in such languages can then be seen as all the *calls* to or *invocations* of these functions, and the constraints on the order in which they may be called.

Large systems often employ a component-based software architecture to manage their complexity. The many components are independent units of development and deployment, encapsulate functionality and allow for re-use [91, 119, 125]. Functions may then be called internally within a component and to communicate between components connected via interfaces, e.g., remote procedure calls.

4.1.2 State machines

We consider software behavior in terms of sequences of discrete *events*, e.g., the start and end of function calls. We define an *alphabet* Σ to be a finite set of events of interest. A *trace* $t \in \Sigma^*$ represents a single finite execution[1], with $*$ the Kleene star. The length of t is denoted by $|t|$ and its i-th event by t_i for $1 \leq i \leq |t|$. An execution *log* is a set of observed traces, and can for instance be obtained by explicit logging or through sniffing tools.

A *state machine* or *automaton* compactly and intuitively represents multiple executions. We define a *Non-deterministic Finite Automaton* (NFA) $A = (S, \Sigma, \Delta, I, F)$ as a 5-tuple, with S a finite set of states, Σ a finite set of events (the alphabet), $\Delta \subseteq S \times \Sigma \times S$ a set of transitions, $I \subseteq S$ a set of *initial states*, and $F \subseteq S$ a set of *accepting states*. *Deterministic Finite Automata* (DFAs) are a sub-class of NFAs allowing for each source state and event only a single target state. An NFA can be determinized to a DFA [50].

A trace $t \in \Sigma^*$ is *accepted* by an NFA $A = (S, \Sigma, \Delta, I, F)$ iff there exists a sequence $(s_0, t_1, s_1), (s_1, t_2, s_2), ..., (s_{|t|-1}, t_{|t|}, s_{|t|}) \in \Delta^*$ with $s_0 \in I$ and $s_{|t|} \in F$. Traces that are not accepted are *rejected*. The *language* $\mathcal{L}(A)$ of an NFA A is the set of all its accepted traces, i.e., $\mathcal{L}(A) = \{t \in \Sigma^* \mid A \text{ accepts } t\}$. The behavior presence predicate $B(A)$ indicates whether A has any behavior, i.e., $B(A) = (\mathcal{L}(A) \neq \emptyset)$. State machines can be *minimized* to a representation with the least number of states possible, while still accepting the same language [64, 103]. Given two NFAs A_1 and A_2, union and intersection are defined as operations that reflect the effect on their resulting languages, i.e., $\mathcal{L}(A_1 \cup A_2) = \mathcal{L}(A_1) \cup \mathcal{L}(A_2)$ and $\mathcal{L}(A_1 \cap A_2) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$, respectively [115].

A (minimal) state machine can be obtained from an execution log through model learning, e.g., using state machine learning algorithms [51, 60, 82] (see Chapter 3) or through active automata learning [60, 68] (see Chapter 2).

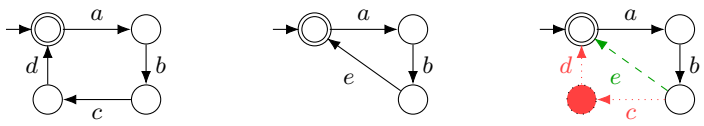
4.1.3 State machine comparison

State machines can be compared in various ways. Walkinshaw and Bogdanov [126] differentiate two perspectives: language-based and structure-based comparisons.

The language perspective considers to which extend the languages of state machines overlap. Two state machines A_1, A_2 are *language equivalent* ($=_L$) iff they accept exactly the same language, i.e., $A_1 =_L A_2 \Leftrightarrow \mathcal{L}(A_1) = \mathcal{L}(A_2)$. A state machine A_1 is related by *language inclusion* ($\leq_L$) to state machine A_2 iff the language of A_1 is included in that of A_2, i.e., $A_1 \leq_L A_2 \Leftrightarrow \mathcal{L}(A_1) \subseteq \mathcal{L}(A_2)$. Various other types of well-known binary equivalence and inclusion relations exist [49], as well as non-binary ones such as precision and recall [116, 126]. We use language equivalence and inclusion as these are commonly used in automata theory, are sufficient to capture the order of function calls, and can be easily explained even to engineers without a formal background. For finite state machines these relations can be computed on their finite structures [36].

Language-based comparison considers the externally observable behavior of state machines. Complementary to it, structure-based comparison considers the overlap of their internal representations in terms of states and transitions.

[1] What is called a 'trace' here, is called a 'word' in Chapter 3.



(a) Source NFA (b) Target NFA (c) Diff NFA

Figure 4.1: Source and target NFAs and their structural differences as a diff NFA.

Walkinshaw and Bogdanov define the *LTSDiff* algorithm [126] that takes two state machines and computes a *diff* state machine: a compact representation of their differences. Figure 4.1 shows an example. A diff state machine is a regular state machine with its states and transitions annotated to represent difference information, i.e., 'unchanged' (black/solid), 'added' (green/dashed) and 'removed' (red/dotted).

The algorithm has three steps:

1. Compute similarity scores for all possible pair-wise combinations of states from the two NFAs being compared. A local score considers only the overlap in directly connected incoming and outgoing transitions of the states. It is extended to a global score by recursively considering all context, using an attenuation factor to ensure closer-by context counts more towards the score than further away context.

2. Use the scores to heuristically compute a matching between states of the two NFAs based on landmarks, a percentage of the highest scoring pairs that score at least some factor better than any other pairs, with a fallback to the initial states. The most obviously similar state pairs are matched first and these are then used to match the surrounding areas, rejecting any remaining conflicting state pairs. The next-best remaining state pair is then selected and matched, etc, until no state pairs are left to consider.

3. Use the matching to compute the diff state machine.

The LTSDiff algorithm has the advantage that it does not require states to be reachable from initial states, does not require state machines to be deterministic or minimal, does not rely on state labels, and that it produces relatively small diffs in practice, unlike some other approaches [76, 99, 105, 117].

For a more elaborate introduction to the LTSDiff algorithm, see Chapter 5. For a more extensive overview of alternative approaches to compare state machine languages and structures, see the work of Walkinshaw and Bogdanov [126] as well as Chapter 5.

4.2 Behavioral comparison methodology

The language and structure-based state machine comparison approaches are complementary. However, to the best of our knowledge there is no work that fully exploits the complementary nature of these approaches, to provide intuitive insights into the behavioral impact of changes for industrial-scale software-intensive

systems. Our methodology takes advantage of their complementary nature in a novel way, to allow handling the complexity of such scale.

As input our methodology takes any number of *model sets* representing, e.g., different software versions, configurations or usage scenarios. They contain state machines that represent behaviors of a number of *entities* representing, e.g., software functions or components. Formally, let E be a finite set of (behavioral) entities and $\mathcal{N}$ the set of all NFAs. A model set $S : E \to \mathcal{N}$ is a complete mapping of entities to models (NFAs). An incomplete mapping can be made complete using $(\emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$ for unmapped entities, thus using an NFA with empty sets of states, events, transitions, and so on. As input our methodology takes a finite set of entities E and a finite set of model sets $\mathbb{S} = \{S_1, ..., S_n\} \subseteq \mathcal{N}^E$.

Figure 4.2 shows the model sets that we use as a running example. Model set S_4 that represents for instance configuration 4, there is no model for entity E_4 that represents for instance (software) function 4. If these models were obtained through model learning on execution logs, no behavior was observed for function 4 using configuration 4.

Our methodology compares the state machines of all input model sets. The results are represented at six levels of abstraction (Figure 4.3). The first three levels focus on model sets and the last three on individual (models of) entities within them. For both model sets and models, the first level considers different behavioral variants, the second level relates the variants, and the third level elaborates on variant differences. Users are guided step by step through the levels, by gradually zooming in on more details, letting them focus on relevant differences. Levels 1 – 5 contain information from the language perspective (L), while levels 5 and 6 contain information from the structural perspective (S). Next, we discuss each of the six levels in more detail.

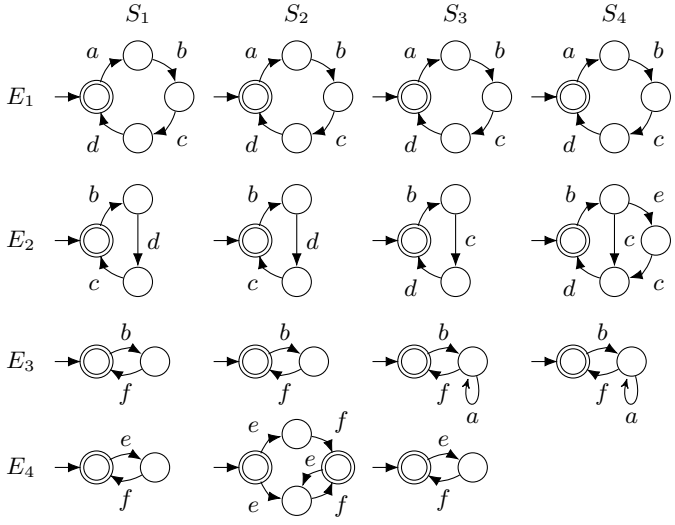


Figure 4.2: The input state machines for the running example, for entities E_1 through E_4 (rows) and model sets S_1 through S_4 (columns). $S_4(E_4) = (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$.

Model sets			Models		
Level 1 Variants	Level 2 Variant relations	Level 3 Variant differences	Level 4 Variants	Level 5 Variant relations	Level 6 Variant differences
L	L	L	L	L / S	S

Figure 4.3: Methodology overview: six levels of detail to inspect comparison results.

4.2.1 Level 1: Model set variants

Level 1 provides the highest level overview. It shows whether model sets have the same behavior, i.e., their entity models are language equivalent. Two model sets $S_i, S_j \in \mathbb{S}$ have the same behavior, denoted $S_i =_L S_j$, iff $\forall_{e \in E}\, S_i(e) =_L S_j(e)$.

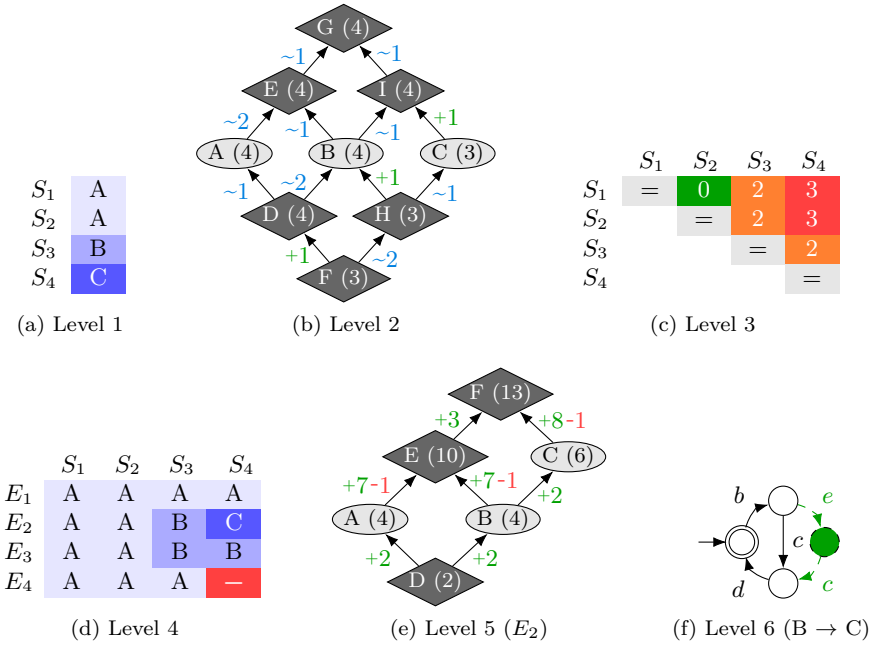
We compare model sets against each other and determine unique model set behavior variants. Variants are formally defined to be equivalence classes of $\mathbb{S}$ under $=_L$, so that $\mathbb{S}/=_L$ is the set of all variants. For presentational clarity we enumerate and refer to different variants of $\mathbb{S}$ in alphabetical order: A, B, etc. We choose a structural representative for each behavioral equivalence class.

Figure 4.4a shows the level 1 result for our running example. Model sets S_1 and S_2 have the same behavior for all four functions and thus get variant A, even though their models for E_4 are structurally different. Model sets S_3 and S_4 get variants B and C as they differ from the other model sets (and each other).

Level 1 thus provides a very high level overview of which model sets have the same or different behavior, and how few or many variants there are. We can see whether this matches our expectations. Depending on the use case, we may be satisfied already after looking at these results. For instance, if we want to know whether different configurations have the same behavior, and if they all have the same variant, we can already conclude that there are no differences in their behavior. If we do go to the other levels, we can ignore model set S_2 as it has the same behavior as S_1. In fact, from the language perspective we can focus on (representatives of) model set variants, each representing one or more models with the same behavior, rather than on individual model sets. Finally, in Figure 4.4a variants are colored using shades of blue like a heat map. In case of many model sets this may reveal patterns, as we will see in Section 4.3.

4.2.2 Level 2: Model set variant relations

Level 1 provides us with model set variants that each have different behavior. Level 2 provides more details. It considers whether the behavior of one model set variant is completely included in the behavior of another variant, i.e., it has less behavior. Formally, for two model sets $S_i, S_j \in \mathbb{S}$, S_i is related to S_j by language inclusion, denoted $S_i \leq_L S_j$, iff $\forall_{e \in E}\, S_i(e) \leq_L S_j(e)$. Given that all model set variants have different behavior, S_i thus has less behavior for at least one entity. Partially ordered set $(\mathbb{S}/=_L, \leq_L)$ can be extended into a finite lattice by computing unions (as supremum) and intersections (as infimum) of representatives of model set variants until a fixed point is reached. The union or intersection of two model



(a) Level 1 (b) Level 2 (c) Level 3

(d) Level 4 (e) Level 5 (E_2) (f) Level 6 (B → C)

Figure 4.4: Behavioral comparison methodology output for the running example: complete levels 1 – 4, level 5 for E_2, and level 6 for E_2 variants B → C.

sets constitutes the per-entity pairwise combination of their entity models, using state machine union or intersection, respectively.

Algorithm 1 shows how a partial lattice of the input model set variants can be computed. As input, a partially ordered set $X = (\mathbb{S}/=_L, \leq_L)$ of level 1 model set variants is given. First, the output lattice (N, E) get initialized, by settings its nodes and edges to empty sets (lines 1 and 2). Then each of the input model set variants $x \in X$ gets considered one by one (lines 3 – 14). First new edges are considered, between x and any existing nodes $n \in N$ that are already in the lattice (lines 4 – 12). To this end, existing ancestor or descendant relations between x and n are considered (line 5). A node $n_1 \in N$ is an ancestor of a node $n_m \in N$, denoted $n_1 <_N n_m$, if and only if there exists a non-empty sequence of edges $e_1 = (n_1 \rightarrow n_2), e_2 = (n_2 \rightarrow n_3), ..., e_{m-1} = (n_{m-1}, n_m)$, with $e_i \in E$ for $1 \leq i \leq m-1$, and with $m > 1$. The node n_m is then also a descendant of node n_1. A node is thus not an ancestor or descendant of itself. If there is no such ancestor or descendant relation between x and n yet in (N, E) (line 5), then it is determined whether such a relation should exist. If x is language included in n (line 6), then a directed edge from x to n, denoted $(x \rightarrow n)$ is added to (N, E) (line 7). If on the other hand n is language included in x (line 8), then instead a directed edge from n to x, denoted $(n \rightarrow x)$ is added to (N, E) (line 9). Adding a directed edge to a lattice is accomplished using Algorithm 2. Once all the edges have been added, x is added as a new node in N (line 13). After the algorithm, partial lattice (N, E) has been constructed, which contains all the input model set

Algorithm 1 Compute a partial lattice for level 2

Input: A partially ordered set $X = (\mathbb{S}/{=_L}, \leq_L)$ of level 1 model set variants.
Output: A partial lattice (N, E), with nodes N and directed edges E.

```
1:  N ← ∅
2:  E ← ∅
3:  for all x ∈ X do
4:      for all n ∈ N do
5:          if x ≰_N n ∧ n ≰_N x then
6:              if x ≤_L n then
7:                  Add edge (x → n) to (N, E).
8:              else if n ≤_L x then
9:                  Add edge (n → x) to (N, E).
10:             end if
11:         end if
12:     end for
13:     N ← N ∪ {x}
14: end for
```

Algorithm 2 Add a directed edge to a lattice

Input: A lattice (N, E), and a directed edge $(s \to t)$ to add to the lattice.
Output: The updated lattice (N, E).

```
1:  S ← {n ∈ N | n <_N s} ∪ {s}
2:  T ← {n ∈ N | t <_N n} ∪ {t}
3:  for all s' ∈ S do
4:      C ← {c ∈ N | ∃_{e∈E} e = (s' → c)}
5:      for all d ∈ C ∩ T do
6:          E ← E \ {(s' → d)}
7:      end for
8:  end for
9:  E ← E ∪ {(s → t)}
```

variants, but is not necessarily a complete lattice just yet.

Algorithm 2 shows how a directed edge $(s \to t)$ can be added to a lattice (N, E). First, certain existing edges are removed to ensure that there will be no two paths in the lattice from s to t (lines 1 – 8). Note that s and t have no direct ancestor or descendant relation yet, as per Algorithm 1. Still, duplicate paths may be created by adding $(s \to t)$ to E. To this end, the set S of all ancestors of s is computed, s included (line 1). Similarly, the set T of all descendants of t is computed, t included (line 2). Each node $s' \in S$ is then considered (lines 3 – 8). For such an s', its direct children c are determined and put into C (line 4). Then for all duplicate targets $d \in C \cap T$ (lines 5 – 7), edge $(s' \to d)$ is removed from E, if it exists (line 6). Once the duplicate edges have been removed, the new edge is added (line 9).

As an example of the application of Algorithm 2, consider Figure 4.5. During the execution of Algorithm 1, a partial lattice is constructed, as shown in Figure 4.5a. A directed edge (n_1, n_3) is to be added to the partial lattice. To prevent

(a) Before addition (b) After addition

Figure 4.5: Example of adding directed edge (n_1, n_3) to a partial lattice.

two paths from n_1 to n_2, edge (n_1, n_2) is removed, before edge (n_1, n_3) is added. In the algorithm, $s = n_1$, $t = n_3$, $S = \{n_1\}$, and $T = \{n_2, n_3\}$. There is only one $s' \in S$, namely $s' = n_1$. The children of n_1 are $C = \{n_2\}$. There is only one $d \in (C \cap T)$, namely $d = \{n_2\}$. Hence, edge $(n_1 \to n_2)$ is removed. Finally, edge $(n_1 \to n_3)$ is added. The result is shown in Figure 4.5b. By removing the direct edge from n_1 to n_2, the new path n_1 to n_3 to n_2 is the only path that remains from n_1 to n_2.

The partial lattice is completed by two executions of Algorithm 3. The first execution starts from the partial lattice constructed using Algorithm 1, and adds extra intersection variants where needed. The second execution starts from the result of the first execution, and adds extra union variants where needed. After both executions, (N, E) is a complete lattice. The algorithm uses a queue Q of still to be considered nodes, which is initialized to the level 1 model set variants X (line 1). All elements of X have been added as nodes to N in Algorithm 1, such that $X \subseteq N$. It then processes the queue as long as it is not empty (lines 2 – 26). It picks an element q from queue Q (line 3), and removes it from Q (line 4). It then considers all possible combinations of q with other level 1 models set variants from X, to in the end consider all combinations of elements from X (lines 5 – 25). Several optimizations could be done here for the selection of $x \in X$. For instance, all elements $x = q$ could be skipped as unions and intersections are reflexive, and therefore don't lead to new variants and thus not to new nodes. If x and q are either ancestors or descendants of each other, they also do not need to be combined, as their union or intersection would either result in x or q, and thus not in new variants/nodes. Finally, each combination of x and q needs to be considered only once, as both union and intersection are commutative. The two nodes q and x are combined using o, so either $q \cup x$ or $q \cap x$ is computed, and the result is x' (line 6). Then first it is considered whether new edges need to be added (lines 7 – 20). An edge between q and x' needs to be added (lines 8 – 12) if x' is a new variant ($x' \notin N$), or if it is an existing variant, but it is not yet related ($q \neq x' \wedge q \not<_N x' \wedge x' \not<_N q$) (line 7). If a union was computed (line 8), then an edge from q to its child x' is added (line 9), while if an intersection was computed (line 10), then an edge to q from its parent x' is added (line 11). Similarly, and edge between x and x' is added, if needed (lines 14 – 20). After adding edges, it is considered whether a new node needs to be added (lines 21 – 24). If x' is a new variant not yet in N (line 21), it is added as a node (line 8) and that node needs to be combined with all inputs as well and is therefore also added to the queue (line 9).

Figure 4.4b shows the level 2 lattice for our running example. The variants

Algorithm 3 Complete a partial lattice for level 2

Input: A partially ordered set $X = (\mathbb{S}/{=_L}, \leq_L)$ of level 1 model set variants, a partial lattice (N, E), and an operation $o \in \{\cup, \cap\}$ to combine two nodes.
Output: The updated lattice (N, E).

$Q \leftarrow X$
while $Q \neq \emptyset$ **do**
  $q \leftarrow$ an element of Q
  $Q \leftarrow Q \setminus \{q\}$
  for all $x \in X$ **do**
    $x' = q \; o \; x$
    if $x' \notin N \; \vee \; (q \neq x' \; \wedge \; q \not\leq_N x' \; \wedge \; x' \not\leq_N q)$ **then**
      if $o = \cup$ **then**
        Add edge $(q \rightarrow x')$ to (N, E).
      else if $o = \cap$ **then**
        Add edge $(x' \rightarrow q)$ to (N, E).
      end if
    end if
    if $x' \notin N \; \vee \; (x \neq x' \; \wedge \; x \not\leq_N x' \; \wedge \; x' \not\leq_N x)$ **then**
      if $o = \cup$ **then**
        Add edge $(x \rightarrow x')$ to (N, E).
      else if $o = \cap$ **then**
        Add edge $(x' \rightarrow x)$ to (N, E).
      end if
    end if
    if $x' \notin N$ **then**
      $N \leftarrow N \cup \{x'\}$
      $Q \leftarrow Q \cup \{x'\}$
    end if
  end for
end while

from level 1 are indicated by ellipses containing the variant and number of entity models that have behavior. The extra variants computed to complete the lattice are indicated by diamonds. They are new behavioral variants, and therefore new letters (compared to the ones already used in level 1) are assigned here, in alphabetical order, to identify the new variants (D, E, F, etc). Arrows indicate directed edges that represent inclusion relations, e.g., the behavior of variant D is included in that of variants A and B (and E, I and G, by transitivity). The arrows are labeled with the number of entities with different present behavior (e.g., ~1) and the number of entities with newly present behavior (e.g., +1). Formally, for model set variants S_i, S_j and $S_i \leq_L S_j$, these are computed by $|\{e \in E \mid B(S_i(e)) \wedge B(S_j(e)) \wedge S_i(e) \neq_L S_j(e)\}|$ and $|\{e \in E \mid \neg B(S_i(e)) \wedge B(S_j(e))\}|$, respectively.

Level 2 provides information on which variants have more or less behavior than other variants, whether variants are closely related (direct arrow) or less closely related (via several arrows), and it has quantitative information on the models

within the model sets by means of the labels on the arrows. As for level 1, we can check whether this conforms to our expectations, or not. For instance, if we compare two software versions and we only added new functionality (e.g., new entities), we would reasonably expect the behavior of the old software version to be included in that of the new software version, and we can check whether that is indeed the case. If this is all that we want to know, we can stop here and we do not need to proceed to level 3.

4.2.3 Level 3: Model set variant differences

Level 2 shows us the quantitative differences between model sets via the arrow labels. However, some model set variants are not directly related by an inclusion arrow (e.g., variants A and B). The number of entities with different behavior between them cannot be determined from the lattice, as simply summing labels (e.g., ~1, +1) could count the same entity multiple times. Level 3 provides more details, showing the number of entities with different behavior between all input model sets. That is, for model sets $S_i, S_j \in \mathbb{S}$ it shows $|\{e \in E \mid S_i(e) \neq_L S_j(e)\}|$.

Figure 4.4c shows the level 3 matrix for our running example. Rows and columns are labeled with the input model sets. Cells indicate the number of entities with different behavior. As language equality is a symmetric and reflexive relation, only the upper-right part of the matrix is filled, and the diagonal is labeled with '=' symbols. As expected, model sets S_1 and S_2 have zero entities with different behavior, as they have the same model set variant. Model sets S_1 (variant A) and S_4 (variant C) have three entities with different behavior.

Level 3 provides more detailed quantitative information. It shows not just whether model sets are different, and how many model sets have differences, but also how different they are. The diagonal is colored gray as it is not relevant. Numbered cells are colored like a heat map based on a gradient from green (no entities with differences) via yellow and orange to red (most entities with differences). In case of many model sets this may again reveal patterns, as we will see in Section 4.3. Similarly to the previous levels, we can check whether all information matches our expectations, and whether we want to proceed to level 4, or not.

4.2.4 Level 4: Model variants

Levels 1 – 3 focus on model sets. Level 4 zooms in even further and considers the (entity) models within the model sets. Similar to how level 1 identifies model set variants, level 4 identifies model variants for each entity. Formally, for an entity $e \in E$, let $\mathbb{S}_e = \{S(e) \mid S \in \mathbb{S}\}$. We consider equivalence classes $\mathbb{S}_e/=_L$ for each $e \in E$ and enumerate and represent them in alphabetical order: A, B, etc. Note that variants are determined per entity and thus variant A of one entity does not necessarily have the same behavior as variant A of another entity.

Figure 4.4d shows the level 4 matrix for our running example. The cells indicate the behavior variant of the model for the corresponding entity (row) in the corresponding model set (column).

Level 4 is the first level to provide details on which entities differ between model sets. This provides a high level overview of the behavior variants for entity models, similar to how level 1 provides it for model sets. We can see the variants,

how many there are, for which models sets, and whether this is expected or not. Depending on the use case, we may again stop at this level if it answers our questions, e.g., in case of checking for regressions if each entity has only a single behavior variant. Otherwise, we can reduce the number of entities to consider for subsequent levels, e.g., skip the ones without regressions (only a single variant, no differences). Furthermore, we may then focus only on unique entity model variants instead of all individual entity models. Finally, the matrix cells are again colored using shades of blue like a heat map. Models without behavior are indicated as a red cell labeled '—' to make them stand out. Here too, in case of many model sets this may reveal patterns, as we will see in Section 4.3.

4.2.5 Level 5: Model variant relations

Level 5 shows relations between entity model variants of level 4, similar to how level 2 shows relations between model set variants of level 1. Formally, for an entity $e \in E$ we have a partially ordered set $(\mathbb{S}_e/=_L, \leq_L)$, which we extend to a finite lattice using unions and intersections, similar to level 2.

Figure 4.4e shows the level 5 lattice for our running example, for entity E_2. We use a representative model for each entity model variant (set of equivalent models). The node shapes and arrows are as in level 2. The node labels now indicate the number of transitions of the model, and the arrow labels indicate the number of added (e.g., +7) and removed transitions (e.g., -1). These are based on the structural comparison that we use and will explain further for level 6. In our example, the behavior of variant B is included in the behavior of variant C.

Level 5 provides information on which entity model variants have more or less behavior, how closely they are related, and the amount of changes between them. As for previous levels, we can check whether this conforms to our expectations, or not. We can also use it to decide what to inspect in more detail in level 6.

4.2.6 Level 6: Model variant differences

Level 6 is the last level. It shows all structural differences between two entity model variants of level 5 as a diff NFA, computed with the LTSDiff algorithm.

Figure 4.4f shows the level 6 diff NFA for our running example, for variants B and C of entity E_2. Variant C (from model set S_4) has two extra transitions in its state machine, and this is clearly visible as two green arrows in this figure.

Level 6 provides the most detailed behavioral differences. Diff NFAs show differences in terms of states and transitions within models. As with the other levels, we can check whether this matches our expectations, or not.

4.3 Evaluation

We perform an empirical evaluation of our methodology through an exploratory field study [110, 118]. To gain some first evidence of both its practical potential and its ability to handle large systems, we perform three case studies at ASML. The first two case studies provide some preliminary evidence of our methodology's practical value, by showing the benefits of all six of its levels, as well as finding a

regression. The third case study shows that our methodology can be applied to a large industrial system, providing insights into its behavior. We have completely automated our approach, in the MIDS tool (see Chapter 6).

ASML develops photolithography systems for the semiconductor industry. These systems process *wafers* (thin circular slices of silicon) in batches (*lots*). Multiple circuits (*dies*) are produced on a single wafer. After the wafer's height profile is *measured*, a light source *exposes* the chip pattern onto a wafer through a projection mask (a *reticle*). A reticle may contain a full-sized pattern (*full field*) or a smaller one (*narrow field*). Computational lithography software uses the *measurements* to compensate for nano-scale imperfections during *exposure*.

In this section the start of function call f is denoted as $f^{\uparrow}$ and its end as $f^{\downarrow}$.

4.3.1 Case study 1: Legacy component technology migration

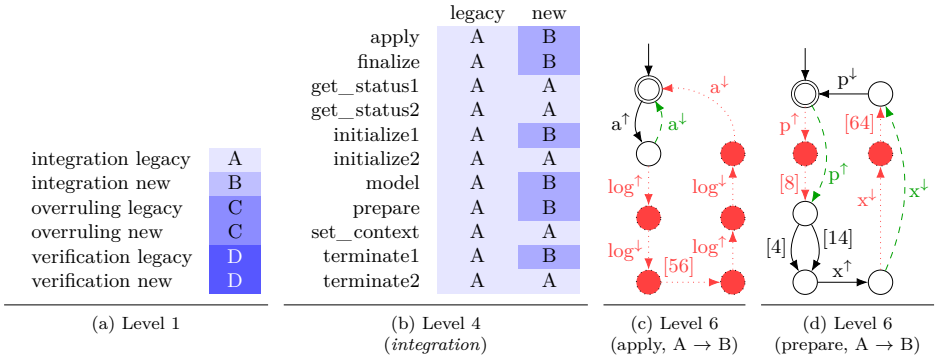
For the first case study, we look at a relatively small computational lithography component, developed and maintained by two engineers. It is internally implemented using legacy end-of-life technology and is migrated to new technology, without changes to its external interface. The engineers thus expect to see the same external behavior in communications with the other components, and we apply our approach to see whether this is indeed the case.

We observe six executions, using three different test sets for both the legacy and new implementations. The *integration* test set contains integration tests. The *overruling* and *verification* test sets each test different configuration options and functionality of the component. Each test set contains multiple tests. For reasons of confidentiality we do not explain them in more detail. For each observed execution, we obtain an execution log capturing the component's runtime communications with other components. The log for each execution is split into separate logs for each of the functions in the component's external interface. Using the CMI state machine learning algorithm (see Chapter 3), we obtain six model sets (one for each execution), with 11 interface functions of the component as entities. The model sets together contain 46 models with behavior, with 2 to 578 states per model, and a sum total of 1,330 states. We run our tool, which takes about 3.38 hours on a standard laptop, mostly spent on executing LTSDiff, and discuss the results per level.

Level 1 (Figure 4.6a): Only for *integration* there are differences in behavior between the legacy and new implementations. As the other two test sets show no differences, they do not need further inspection. Given that we then have only two model sets left, we skip levels 2 and 3, and proceed directly to level 4.

Level 4 (Figure 4.6b): We see the 11 functions, anonymized for confidentiality reasons, and their behavioral variants. Only 6 out of 11 entities show differences in behavior, to be inspected in more detail. Given that they all have only two variants per entity, we skip level 5 and proceed directly to level 6.

Level 6 (Figures 4.6c and 4.6d): Figure 4.6c shows the diff NFA for function 'apply' (abbreviated to 'a'), for variant A to variant B. The figure shows that the new implementation involves only the start and end of this function. The legacy implementation has more behavior, as within the 'apply' function it has 30 calls



(a) Level 1 (b) Level 4 (*integration*) (c) Level 6 (apply, A → B) (d) Level 6 (prepare, A → B)

Figure 4.6: First results for case study 1: complete level 1, level 4 for the *integration* test set, and level 6 with variants A vs B for functions 'apply' and 'prepare'.

(with returns) to a 'log' function. In the figure, only the first and last of these calls (with their returns) are shown, and the remaining sequence of 56 transitions, representing 28 calls and their returns, is abbreviated to '[56]'. Figure 4.6d shows the diff NFA for function 'prepare' (abbreviated to 'p'), for variant A to variant B. For reasons of confidentiality and presentational clarity again several sequences of transitions are abbreviated. Here, the figure shows that the legacy implementation invokes the 'log' function 4 and 32 times, indicated as '[8]' and '[64]', respectively, while the new implementation does not.

Having inspected the differences for only two entities, it appears that all 'log' function calls are missing in the new implementation. The component engineers confirmed that indeed for the new implementation the component was not yet hooked up to the logging framework. Our approach clearly shows this regression.

To look for other differences in behavior, we remove all 'log' function calls and returns from the models of the legacy implementation. To do so, we rename all 'log' function call and return events to ε and apply weak-language normalization [115]. We run our tool again, which now only takes a mere 19 seconds.

Level 1 (Figure 4.7): Looking at the new results for level 1, we immediately see that there are no more observed differences in behavior for the legacy and new implementations, for all three test sets. We do not see any further regressions in behavior, and we thus do not have to go to further levels.

Figure 4.7: New results for case study 1: level 1.

Given that the engineers consider this component to have quite a good test set with adequate coverage, our approach is applied as an extra safety net that complements traditional testing, akin to differential testing [54]. As any change in the (order of) communications with other components will show up in our models and comparisons, it is like having assertions for all external communications. Both engineers find this valuable. They would like to apply our methodology also for larger and more complex technology migrations, where they foresee even more value.

4.3.2 Case study 2: Test coverage

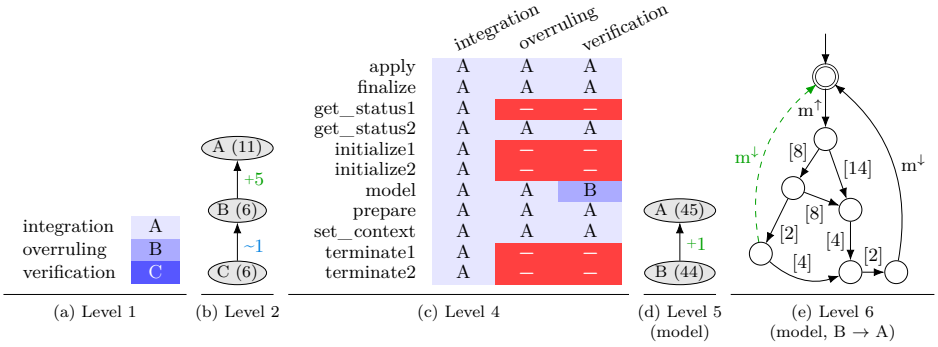
The second case study considers again the same component and three test sets from the first case study, but from a difference angle. Instead of comparing the legacy and new implementation, we compare the three test sets against each other. The goal is to see how the behaviors of the components that communicate during the different test sets differ, and whether one or more test sets are perhaps superfluous. We use the versions of the input models, and results of running our tool, from the first case study where the 'log' function is completely removed. We discuss the results of applying our methodology, per level:

Level 1 (Figure 4.8a): The three tests sets have different behavior (A – C).

Level 2 (Figure 4.8b): The *integration* test set (variant A) has behavior for all 11 functions, and the other two test sets (B, C) for 5 fewer functions, i.e., 6 functions. Also, *integration* (A) includes all the behavior of the other two test sets, while *verification* (C) differs from *overruling* (B) by only one function. As all variants are (transitively) related in the lattice, we skip level 3.

Level 4 (Figure 4.8c): We clearly see which 5 functions are only used during the *integration* tests. The component engineers expect this difference, as for *overruling* and *verification* these 5 functions are stubbed internally and are thus not externally visible. Also, for the *verification* tests only the 'model' function has different behavior. We inspect this further in level 5.

Level 5 (Figure 4.8d): The behavior of the 'model' function for variant B



(a) Level 1 (b) Level 2 (c) Level 4 (d) Level 5 (model) (e) Level 6 (model, B → A)

Figure 4.8: Results for case study 2: complete levels 1, 2 and 4, level 5 for function 'model', and level 6 for function 'model' variants B vs A.

(*verification*) is included in that of variant A (*integration* and *overruling*), which has one additional transition. We inspect this further in level 6.

Level 6 (Figure 4.8e): Here we see the diff NFA for function 'model' (abbreviated to 'm'), for variant B to variant A, following the arrow in the level 5 lattice. For confidentiality and presentational clarity we annotate other arrows with $[n]$ to abbreviate n transitions in sequence. The one extra transition of function variant A is clearly visible. There it is possible to return to the initial state earlier on, skipping part of the behavior of the state machine. The engineers again expect this, as some functionality is not activated depending on the component configuration.

The comparison results suggest that since the *integration* test set covers more behavior than the other two test sets, those other two test sets can be removed. This would be a valid conclusion, if one only considers function call order, as we do for our methodology. However, functions could have different behavior for different arguments. If this results in a difference in which functions are called or in what order they are called, then our approach will highlight such differences. If however for different configurations there are differences in which paths through a state machine are taken for which argument values, while each path is still taken for some argument value, this would not be visible with our current approach. The different test sets that we consider do indeed test different configurations using different argument values, and hence they do add value and can not simply be removed. Fully taking the influence of argument values into account is considered future work (see also Section 7.2.3).

In any regard, our methodology provides insight into the behavioral differences for the various configurations and functional scenarios considered by the different test sets. This can be automatically obtained even by engineers who are not domain experts.

4.3.3 Case study 3: System behavior matching recipe

For the third case study, we investigate how *recipes* containing information on the number of wafers and used reticles relate to the system behavior. ASML's customers can specify their own recipes to configure their lithography systems for their purposes, e.g., to create CPU or memory chips. The software running on the systems will exhibit different behavior for different recipes, and thus software behavior offers a lens to look at system behavior.

Table 4.1 shows the recipes that we consider for this case study. For reasons of confidentiality, we do not explain the origin of these recipes and we consider only the details relevant for this case study. There are six lots, each with their own recipe. Lots 1 and 2 have five wafers each and the other lots have 15 wafers each. There are two reticles, X and Y. For lot 1, reticle X is used 96 times, one for each

	Lot 1	**Lot 2**	**Lot 3**	**Lot 4**	**Lot 5**	**Lot 6**
Wafers	5	5	15	15	15	15
Reticle	96*X	96*Y	96*X	96*Y	124*X, 1*Y	125*X
Field	Full	Full	Full	Full	Narrow	Narrow

Table 4.1: Case study 3: recipes for the different lots.

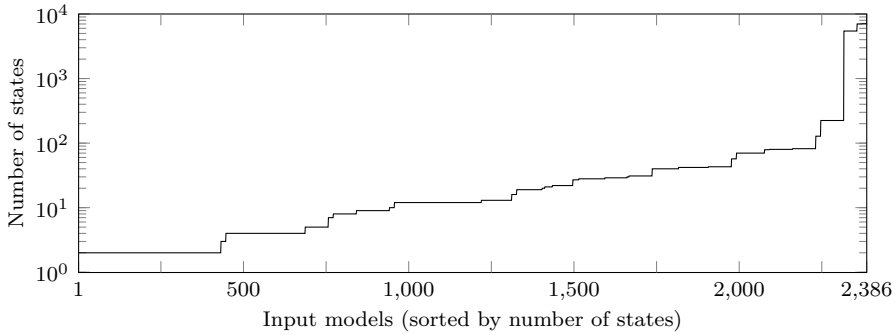


Figure 4.9: Case study 3: sizes of the input models with behavior.

die. Lot 5 uses both reticles. Exposure can be done using full field or narrow field, where narrow field leads to more exposures (125 rather than 96).

We consider the behavior of the exposure sub-system, i.e., 32 software components involved in the high-level exposure control. Observing the system execution for about an hour as it initializes and processes lots, we obtain a single execution log capturing all observed inter-component communications. This log is split into multiple logs, one for each of the 85 exposures (one per wafer and for lot 5 twice per wafer as it uses two reticles). The exposure logs are further split into separate logs for each of the components, containing only their interactions with the other components. Using the CMI state machine learning algorithm (see Chapter 3), we obtain 85 model sets (one per exposure), containing models of the 32 components (entities). Model sets may lack a certain component model if that component did not interact with other components during the corresponding exposure. Figure 4.9 shows the sizes of the input models in number of states. The 85 model sets together contain 2,386 models with behavior, with 2 to 7,070 states per model, and a sum total of 495,505 states, making this a large case study.

We run our tool, skipping levels 2 and 5 as they are less relevant for this case study. For LTSDiff, local instead of global scoring is used when state machines with more than 100 states are involved, sacrificing accuracy for performance. Running the tool takes about 1.23 hours. We discuss the results per level.

Level 1 (Figure 4.10): We discuss multiple observations based on patterns that are visible in level 1. Each exposure is indicated with an identifier, such as 3-10 for the exposure of wafer 10 from lot 3, and 5-15B for the second exposure (B) of wafer 15 from lot 5. Different gradient colors are used for the different behavioral variants, for presentational clarity.

a) *First exposure of a lot*: For lots 1 – 4, the main behavior variant is variant B. The first exposures of these lots however all have different behavior than B, namely A or D.

b) *Changes during a lot*: For lots 2 – 4 we also see different behavior for some exposures later during the lot (C, E).

c) *Reticle swaps*: All exposures of lot 5 (F – L) have behavior different than the

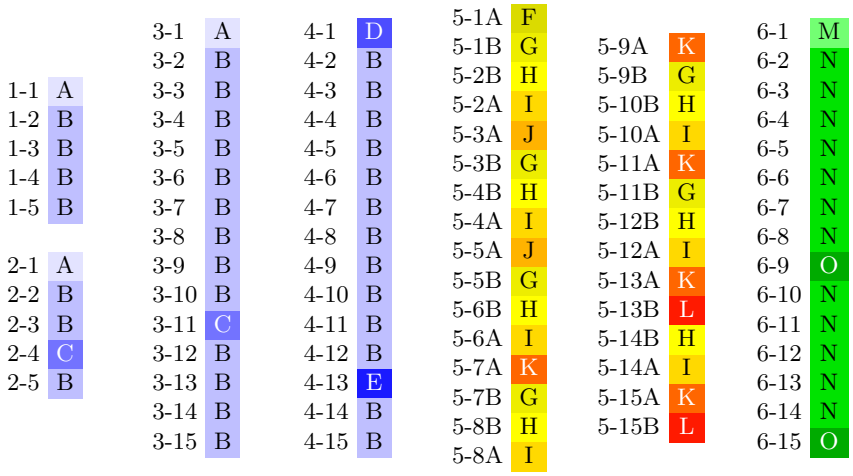


Figure 4.10: Results for case study 3: level 1.

other lots (A – E, M – O). Lot 5 is the only lot where two reticles are used per wafer, and thus reticles must be swapped regularly. To minimize the number of swaps, the system uses an 'XYYX' pattern for every two wafers (first wafer reticle 'X', first wafer reticle 'Y', second wafer reticle 'Y', second wafer reticle 'X'). These patterns of four exposures are clearly visible in the model set variants (J – G – H – I, K – G – H – I).

d) *Full field vs narrow field*: The difference between lots 1 and 3 compared to lot 6 is the use of full vs narrow field. The behavior for lots 1 and 3 (A – C) and lot 6 (M – O) differ, but they have similar structure (mostly the same variant, first exposure and some exposures during the lot are different).

Level 3 (Figure 4.11): We elaborate on each of the four observations using the results for level 3.

a) *First exposure of a lot*: For lots 1 – 4, we mainly see regular behavior (dark green, no components with different behavior). For the first exposures of these lots we do see differences (yellow lines, mainly 2 or 3 components).

b) *Changes during a lot*: For lots 2 – 4 we again see differences for some exposures later during the lot (light green and yellow lines, mainly 1 or 2 components).

c) *Reticle swaps*: The reticle swaps are again very much visible for lot 5 (vertical orange, red and light green lines in a repeating pattern of 4 columns).

d) *Full field vs narrow field*: Observe the differences between thick-border enclosed areas left and right of the figure. These full field (lots 1 + 3) vs narrow field (lot 6) differences seem to be caused by a single component.

Level 4 (Figure 4.12): The observations are detailed even further using the results for level 4.

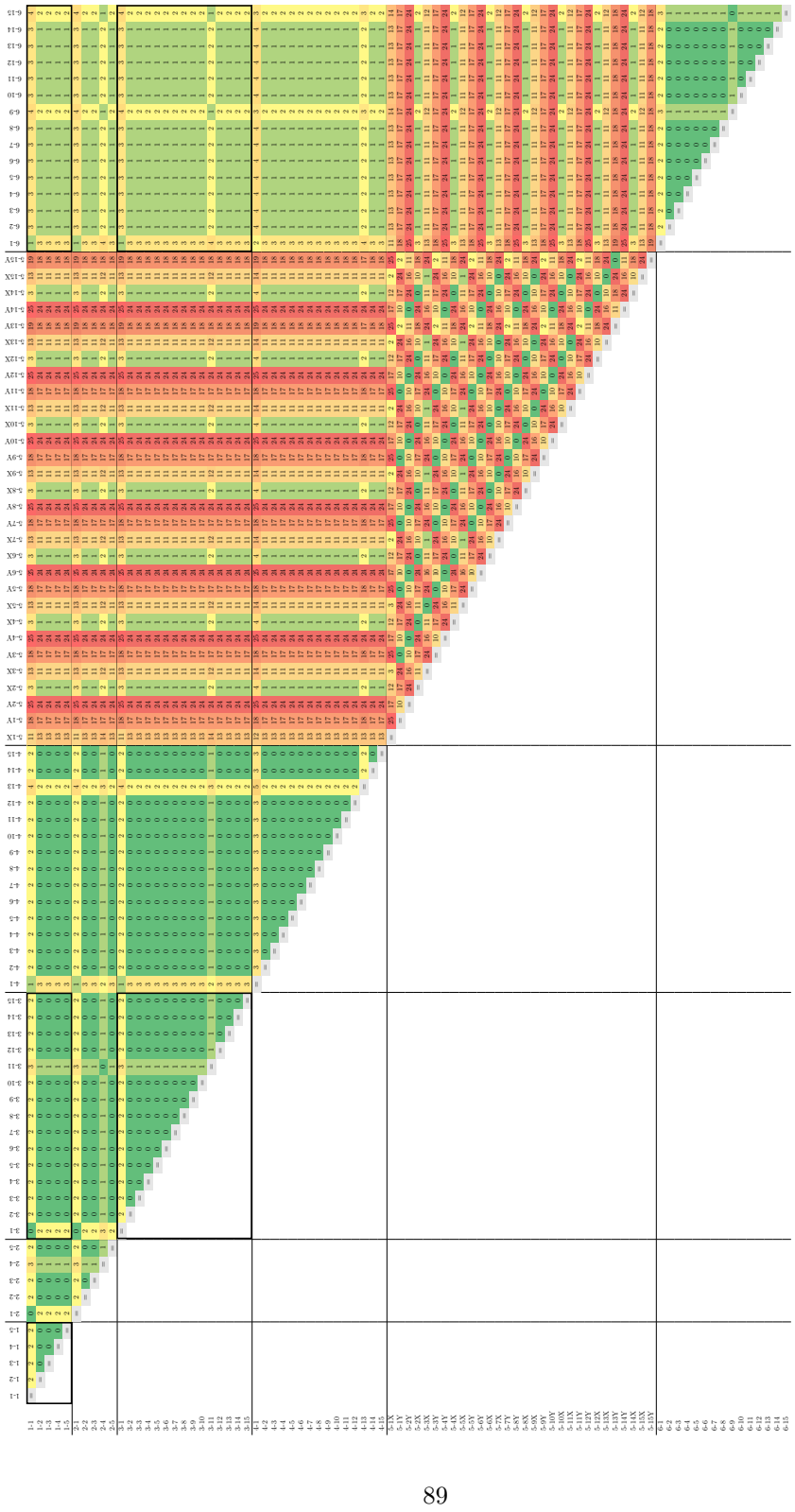


Figure 4.11: Results for case study 3: level 3.

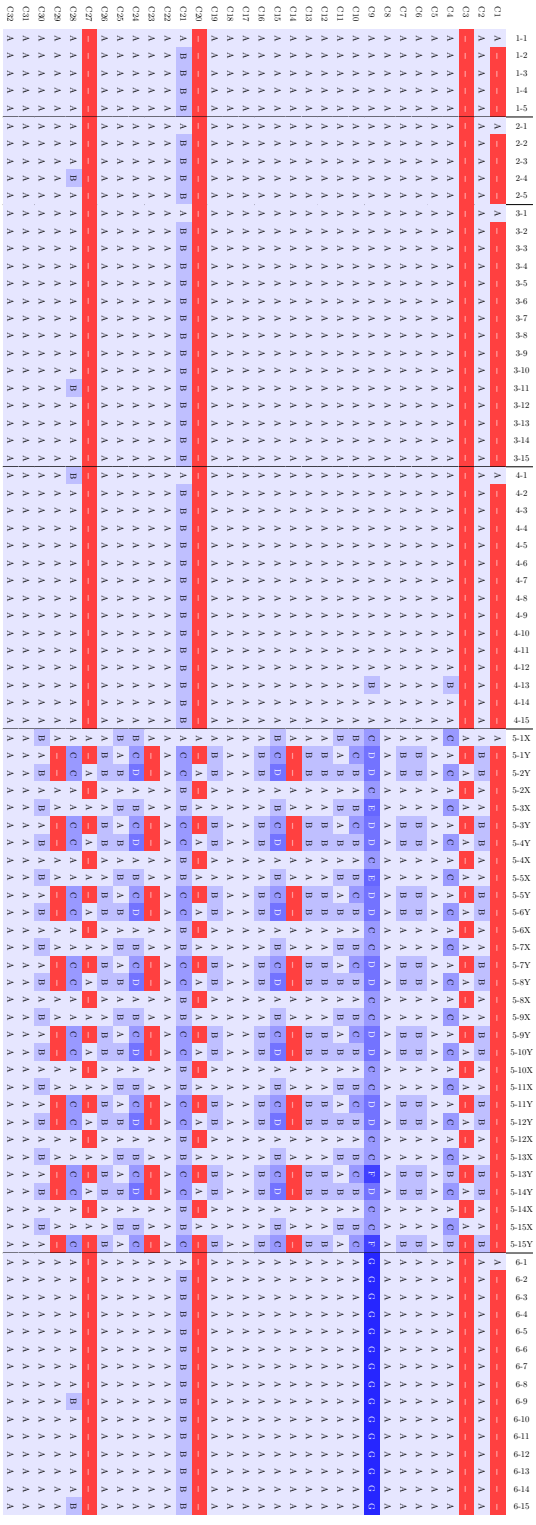


Figure 4.12: Results for case study 3: level 4. The component names have been anonymized for confidentiality reasons.

a) *First exposure of a lot*: The differences in first exposures of lots 1 – 4 can be attributed primarily to components C1 and C21, and for lot 4 also to C28.

b) *Changes during a lot*: The changes for exposures during lots 2 – 4 can be attributed to components C4, C9 and C28.

c) *Reticle swaps*: The reticle swap differences affect many components. For several components (e.g., C2, C6, C9) we again see the 'XYYX' reticle swap pattern. For some other components (e.g., C3, C4) we see a 'VWVW' pattern instead, relating to first vs second exposure of a wafer.

d) *Full field vs narrow field*: Indeed only one component (C9) causes the full field (lots 1 + 3) vs narrow field (lot 6) differences (variants A/B vs G).

Level 6 (Figure 4.13): For reasons of confidentiality, we focus only on the *first exposure of a lot* differences. We inspect level 6 for variants A and B of component C21. Figure 4.13 shows a part of the diff state machine, with 'l' a logging function, 'i' a function to get some information, and 'q' a query function. For confidentiality reasons we do not explain the functions in more detail. The upper and lower paths indicate that both versions can skip the calls to 'q'. The only difference is that variant A (first wafer, in red) calls 'i' before calling 'q', while variant B (other wafers, in green) does not. The company's domain experts are well aware of such 'first wafer effects'.

Conclusion: The system behavior differs between wafers, and by going through the levels of our methodology we obtain progressive insights into these behavioral differences and how they relate to the recipes. This allows engineers to understand how different configurations influence the system behavior, e.g., which components are affected by reticle swaps or full field vs narrow field, and in what way they behave differently. While the input contains a large number of state machines, with an even larger number of states, our methodology allows engineers to step by step zoom in on parts of this behavior, thus making it suitable to analyze this large system.

Our approach has many potential applications. For instance, understanding how certain configurations affect the system behavior is key when changing the system behavior. Junior engineers can understand the system and its configurations without having to rely on domain experts. Domain experts can check whether their mental views conform to reality, and adapt their mental views if they turn out to be outdated or incomplete. Furthermore, if certain configurations have no

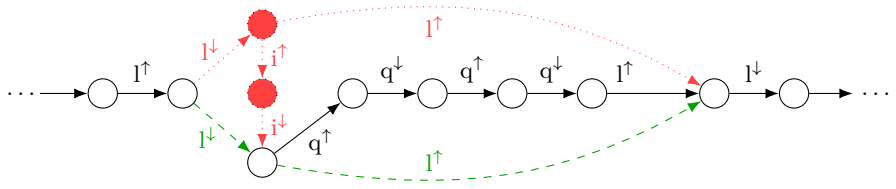


Figure 4.13: Results for case study 3: excerpt of level 6 (C21, A → B).

effect at all on the system behavior, they could be removed from the system to avoid having to consider them when changing the system.

4.4 Conclusions and future work

We contribute a novel multi-level methodology for behavioral comparison of software-intensive systems. It integrates multiple existing complementary methods to automatically compare the behavior of state machines. Our methodology takes advantage of their complementary nature in a novel way, using six levels with progressive detail to handle the complexity of large industrial systems.

Our qualitative exploratory field study suggests that our approach allows one to inspect the behavioral differences of large systems, and that it has practical value for getting insight into system behavior for various configurations and scenarios, and preventing regressions. However, a more rigorous and quantitative evaluation of our methodology is still needed (see Chapter 5).

Our work is generically applicable as it works on state machines, which are widely used and understood in both computer science and industry. We plan to research the generality of our approach by also applying it at other companies with software-intensive systems that have suitable state machine models [21] (see Section 6.2.2). We also plan to make the MIDS tool, which includes our multi-level comparison approach, publicly available as an open-source tool (see Chapter 6).

Other future work includes extensions beyond comparing NFAs to consider also Extended Finite Automata and Timed Automata as input to our approach (see also Section 7.2.1), and adding actionable insights beyond merely behavioral differences to further support change impact analysis (see [7]). Our methodology could also be applied to different use cases such as diagnosis of unstable tests and field issues.

Chapter 5

gLTSdiff: Generalized and extended structural comparison

Various kinds of state machine models exist, such as labeled transition systems, (extended) finite automata, feature finite state machines, and timed automata. They are broadly used in academia and industry alike, to model software behavior, as well as for other purposes. In practice, state machine models are often compared, for instance to evaluate the accuracy of different model learning algorithms [105, 126], to find regressions in new software versions before they are deployed [132] (see also Chapter 4), and to perform software fingerprint matching for security applications such as malware detection, copyright infringement, vulnerability analysis, and digital forensics [9, 41].

Methods to compare state machine models can be divided into behavior-based methods, well-suited for comparing many state machines and providing a broad overview, and structure-based methods, which make detailed differences tangible [126] (see also Chapter 4). Various structural model comparison approaches have been proposed in the literature [99, 105]. State of the art is the LTSDiff algorithm by Walkinshaw and Bogdanov [126]. It is fully automated and has limited assumptions, making it broadly applicable. However, its limited assumptions also mean that additional information of specific state machine model representations is not taken into account, requiring adaptations to prevent sub-optimal or even invalid results [23, 39, 40, 99].

We introduce gLTSdiff, which generalizes and extends LTSDiff. Our main contribution is gLTSdiff's generality. It supports a wide range of state machine model representations, taking their additional information into account by matching over the (fine-grained) structure of state and transition labels. And it can be configured to support additional representations. We also discuss further challenges we faced while applying LTSDiff in an industrial context. gLTSdiff addresses them, by rewriting undesired difference patterns, supporting comparison of any number of input models, and allowing for an effort/quality trade-off.

The gLTSdiff approach is implemented as an extensible open-source library.

We apply it to several large-scale industrial and open-source case studies, to show that gLTSdiff provides an extensible, configurable, scalable approach to compare various kinds of state machine models, supporting a variety of real-world applications. Concretely, we show its practical value, that it handles large numbers of input models, reduces the number of differences in comparison results, and allows for an effort/quality trade-off. For reproducibility, all input models, code and comparison results of the evaluation are available as a public artifact [57].

We discuss related literature and the LTSDiff algorithm in Section 5.1, the challenges in applying it in Section 5.2, our generalized and extended approach gLTSdiff in Section 5.3, and our open-source library and its design in Section 5.4. We evaluate our approach in Section 5.5, before concluding in Section 5.6.

5.1 Background

5.1.1 General notations

The length of a sequence or tuple t is denoted by $|t|$, its i-th element by t_i or $t[i]$ for $1 \leq i \leq |t|$. We further denote the symmetric difference of two sets by $\ominus$, and the powerset of set X by $\mathcal{P}(\mathrm{X})$. The domain $\mathsf{dom}(f)$ and range $\mathsf{rng}(f)$ of a partial function $f : X \rightharpoonup Y$ are defined as $\{x \in X \mid f(x) \text{ is defined}\}$ and $\{f(x) \mid x \in \mathsf{dom}(f)\}$, respectively, with $\mathsf{dom}(f) \subseteq X$ and $\mathsf{rng}(f) \subseteq Y$.

5.1.2 State machines

State machines can be used to model (software) behavior, for instance the order of function calls and returns. State machines exist in various flavors. We define a finite *Labeled Transition System* (LTS), simply called LTS from here on, and a *Non-deterministic Finite Automaton* (NFA) as usual:

Definition 5.1 (LTS)**.** *An LTS $L = (S, \Sigma, \Delta, I)$ is a 4-tuple with S a finite set of states, Σ a finite set of events or labels, $\Delta \subseteq S \times \Sigma \times S$ a set of transitions, and $I \subseteq S$ its initial states.*

Definition 5.2 (NFA)**.** *An NFA $A = (S, \Sigma, \Delta, I, F)$ is a 5-tuple with (S, Σ, Δ, I) an LTS, and $F \subseteq S$ a set of accepting or final states.*

Deterministic Finite Automata (DFAs) are a sub-class of NFAs allowing for each source state and event at most one target state. We use subscripts to distinguish multiple state machines, such as two LTSs $L_1 = (S_1, \Sigma_1, \Delta_1, I_1)$ and $L_2 = (S_2, \Sigma_2, \Delta_2, I_2)$. We further use superscripts to refer to elements of state machines. For example, L_1^S and L_2^S refer to the sets of states of L_1 and L_2, respectively. For a transition $t = (s, \sigma, s') \in \Delta$, we define $t^{src} = s$, $t^{lbl} = \sigma$ and $t^{tgt} = s'$, to refer to its source state, label and target state, respectively.

5.1.3 State machine comparison

Rather than general model comparison methods [29, 75], we consider ones specific to state machines. Walkinshaw and Bogdanov [126] differentiate behavior-based

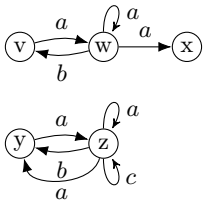
and structure-based methods. Behavior-based methods consider the externally observable behavior of state machines in terms of sequences of events. They include qualitative equivalence and inclusion relations such as bisimulation equivalence and language inclusion [49], and quantitative measures such as precision and recall [116, 126]. Complementary, structure-based methods consider the overlap of their internal model representations in terms of their states and transitions. Both are useful: behavior-based methods are well-suited for comparing many state machines and providing a broad overview, while structure-based methods allow revealing their detailed differences (see Chapter 4).

Various algorithms for structural comparison of state machines exist. Quante and Koschke [105] automatically compare two minimal DFAs by computing their union, and from the initial states walking through each input DFA in parallel with the union, finding deleted and inserted transitions along the way. Nejati et al. [99] match hierarchical StateChart models, considering states to match if a weighted average of some typographic, linguistic, depth and structure measures exceeds a certain threshold. Structurally, they consider immediate neighbors for local similarity scores, and take into account further-away states by aggregating scores forwards and backwards through a bounded iterative refinement. Initial values can be customized, and domain experts must review matches. For sound matches, where initial states are matched and behavior and hierarchy are preserved, the input StateCharts are merged into a single parameterized model. Walkinshaw and Bogdanov define the LTSDiff algorithm [126]. LTSDiff is similar to the work of Nejati et al. in various ways, as it computes global similarity scores between states of two input LTSs, heuristically matches states based on those scores, and uses the matching to construct a difference LTS, a merged model of the differences (see Figure 5.1d for an example).

LTSDiff has the advantage that it does not require state machines to be deterministic or minimal, does not require states to be reachable from initial states, does not rely on state labels, does not require input from domain experts, and produces relatively small diffs in practice. The limited assumptions are however a double-edged sword, on the one hand leading to a more universally-applicable algorithm, and on the other hand not allowing to take state information and domain knowledge into account to improve matches. LTSDiff can however be adapted to support specific kinds of models. For example, it was adapted to compare Feature Finite State Machines (FFSMs) [39, 40], and to compare LTSs that distinguish unknown from invalid behavior [23]. LTSDiff can thus be seen as state-of-the-art in structural comparison of software behavior, and serves as a good basis for adaptation and extension towards other kinds of state machine models.

5.1.4 The LTSDiff algorithm

In this chapter, we extend and generalize LTSDiff to gLTSdiff. We briefly discuss the definition of the LTSDiff algorithm by Walkinshaw and Bogdanov [126], using slightly different notation, and illustrate it using the example of Figure 5.1. The structural comparison of two LTSs L_1 and L_2 (Figure 5.1a) involves determining which of their states and transitions are similar and can thus be matched, and which (remaining) states and transitions have then been added or removed to turn L_1 into L_2. LTSDiff assumes no particular knowledge about the states, such



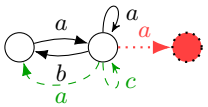
(a) Input LTSs

	vy	vz	wy	wz	xy	xz	
vy	2			$-k$			1
vz		8	$-k$	$-k$			2
wy			6	$-k$		$-k$	2
wz	$-k$		$-k$	$12-k$	$-k$	$-k$	5
xy					2		0
xz						6	0

(b) Linear equations for sc^{glob}_{succ}

Pair	$\mathbf{sc^{glob}_{succ}}$	$\mathbf{sc^{glob}_{pred}}$	$\mathbf{sc^{glob}}$
vy	0.648	0.321	0.484
vz	0.316	0	0.158
wy	0.383	0.380	0.381
wz	0.493	0.470	0.482
xy	0	0.321	0.160
xz	0	0.418	0.209

(c) (Rounded) scores for $k = 0.6$



(d) Diff LTS

Figure 5.1: LTSDiff example, inspired by Figure 3 and Table V of [126].

as state labels. The algorithm matches states based on the similarity of their surroundings, which is captured in *(state) similarity scores*.

1) Similarity Scores: LTSDiff first computes similarity scores for all possible pair-wise combinations of states from L_1 and L_2. A *local* score only considers overlap in directly connected incoming and outgoing transitions of the states. For a state $s \in S$, its outgoing events are defined as $out(s) = \{t^{lbl} \mid t \in \Delta \wedge t^{src} = s\}$. For two states $s_1 \in L_1^S$ and $s_2 \in L_2^S$ their set of common successors contains the pairs of states reachable by common outgoing events:

$$succ(s_1, s_2) = \{(s_1', s_2', \sigma) \mid (s_1, \sigma, s_1') \in L_1^\Delta \wedge (s_2, \sigma, s_2') \in L_2^\Delta\} \tag{5.1}$$

The local successor similarity score is then defined as the total number of common transitions as a fraction of the total number of transitions that constitute possible matches (common and uncommon ones):

$$sc^{local}_{succ}(s_1, s_2) = \frac{|succ(s_1, s_2)|}{|out(s_1) \ominus out(s_2)| + |succ(s_1, s_2)|}$$

In case of division by zero, the score is zero. The local predecessor similarity score sc^{local}_{pred} is computed similarly. For the two example LTSs of Figure 5.1a, $succ(w, z) = \{(w, z, a), (x, z, a), (w, y, a), (x, y, a), (v, y, b)\}$, $out(w) = \{a, b\}$, $out(z) = \{a, b, c\}$ and $sc^{local}_{succ}(w, z) = \frac{5}{1+5}$. The local successor similarity score is extended to a *global* successor similarity score by recursively considering all context:

$$sc^{glob}_{succ}(s_1, s_2) = \frac{1}{2} \frac{\sum_{(s_1', s_2', \sigma) \in succ(s_1, s_2)} \left(1 + k \cdot sc^{glob}_{succ}(s_1', s_2')\right)}{|out(s_1) \ominus out(s_2)| + |succ(s_1, s_2)|} \tag{5.2}$$

Successor count $|succ(s_1, s_2)|$ is replaced by a summation over all successors. Each successor is still counted once, by adding 1, and additionally the global score of

each successor is taken into account, by adding $sc^{glob}_{succ}(s'_1, s'_2)$. Attenuation or discounting factor $0 \leq k \leq 1$ ensures that nearby states contribute more to the score than further-away ones. The $\frac{1}{2}$ fraction in front ensures that $sc^{glob}_{succ} \leq 1$. In case of division by zero, the score is zero instead.

The definition of sc^{glob}_{succ} forms a system of linear equations, by considering $sc^{glob}_{succ}(s_1, s_2)$ as a variable for each state pair $(s_1, s_2) \in L^S_1 \times L^S_2$, and pre-calculating the local knowledge of Equation 5.2, leaving only recursive uses of sc^{glob}_{succ} as unknown variables. For the example, the linear equations to compute sc^{glob}_{succ} are shown in Figure 5.1b. For instance, for variable vy denoting $sc^{glob}_{succ}(v, y)$, using Equation 5.2 we get $vy = \frac{1}{2}\frac{1+k \cdot wz}{0+0+1}$. By multiplying both sides by 2, and simplifying the divisor, we get $2 \cdot vy = \frac{1+k \cdot wz}{1}$. We can then simplify the fraction, to get $2 \cdot vy = 1 + k \cdot wz$. By on both sides subtracting the right-side term with the variable $(k \cdot wz)$, we get $2 \cdot vy - k \cdot wz = 1$. This way, only a constant remains on the right side. This matches the first row in Figure 5.1b. The first column indicates the variable that we are considering (vy). The last column indicates the right side value (1). The other columns indicate the left side of the equation: for $2 \cdot vy$ we get a 2 in the vy column, and for $-k \cdot wz$ we get $-k$ in the wz column.

Solving the system of equations produces the global successor similarity score for each state pair. The global predecessor similarity score sc^{glob}_{pred} can be similarly computed. They are averaged to compute the final global similarity score:

$$sc^{glob}(s_1, s_2) = \frac{sc^{glob}_{pred}(s_1, s_2) + sc^{glob}_{succ}(s_1, s_2)}{2} \tag{5.3}$$

The final scores for the example are shown in Figure 5.1c.

2) State Matching: LTSDiff uses the similarity scores to heuristically compute a matching between states of the two LTSs. It does so based on landmarks, a percentage of the highest scoring pairs (threshold t) that score at least some factor better (ratio r) than any other pairs, with a fallback to the initial states. The most obviously similar state pairs are matched first and are then used to match the surrounding areas, rejecting any remaining conflicting state pairs. Surrounding areas are matched both forwards and backwards. In case there are multiple potential choices, such as non-deterministic choices of transitions, scores are used to select best matches. Once the surrounding areas are matched as far outwards as is possible, the next-best remaining state pair is selected and matched, and so on, until no state pairs are left to consider. The details are not so relevant for our work, but can be found in Algorithm 1 of the LTSDiff paper [126].

For the example, we use $t = 0.25$ and $r = 1.5$. We thus select out of the six possible state pairs the top 25%, namely 'vy' with score 0.484, and 'wz' with score 0.482. For 'vy', r is not relevant, as there are no other selected state pairs with states 'v' or 'y'. Similarly, for 'wz' also r is not relevant, as there are no other selected state pairs with states 'w' or 'z'. Hence, both state pairs are selected as landmarks. The landmarks have no states in common, and therefore do not conflict. Both landmarks thus become matches. No surrounding area is matched, as for the only remaining state 'x' there are no states left in the other LTS against which to match it. State 'x' thus remains unmatched.

3) Diff LTS: Finally, the state matching is used to compute a patch consisting of added and removed transitions, and renamed states. Walkinshaw and Bogdanov

visualize the patch as a Diff LTS, an LTS with its states and transitions annotated to represent difference information, i.e., 'unchanged' (black/solid), 'added' (green/dashed) and 'removed' (red/dotted). We omit patches and use Diff LTSs directly, being sufficient for our work. Patches can however be derived from Diff LTSs.

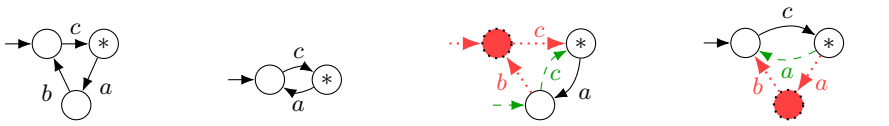
The Diff LTS for the example is shown in Figure 5.1d. The left state corresponds to matched state pair 'vy', the middle state to matched state pair 'wz', and the right state to the unmatched state 'x'. The outgoing a transition from 'v' to 'w' in the first LTS has a matching transition from 'y' to 'z' in the second LTS. Since the source states of these two transitions are matched ('v' and 'y'), as well as their target states ('w' and 'z'), we see in the Diff LTS an unchanged transition labeled with a from the left state to the middle state. Similarly, there is an unchanged transition labeled with b in the opposite direction, and a self loop on the middle state. The transition labeled with a to state 'x' from the first LTS has no counterpart in the second LTS, as state 'x' is unmatched. The transition in the Diff LTS labeled with a from the middle state to the right state is therefore considered a removed transition. Similarly, there are two added transitions.

5.2 Challenges in applying LTSDiff

While applying LTSDiff to various case studies, some of them described in Sections 4.3 and 5.5, we identified several challenges. We briefly describe these nine challenges, as well as how gLTSdiff addresses them.

1) Kinds of Models: The LTSDiff algorithm compares LTSs, others compare FFSMs, and we compare NFAs. Rather than developing yet another algorithm for a specific kind of state machine model, gLTSdiff generalizes LTSDiff to allow structural comparison for a wide range of state machine formalisms.

2) Initial States: The LTSDiff algorithm considers the surrounding network of transitions to determine state matches, where all states are considered in an equal manner. In the example of Figure 5.2, the pair of states marked with '$*$' (Figures 5.2a and 5.2b) gets the highest score and these two states are matched first. Then either of the two remaining states from the source LTS can be matched to the single remaining state of the target LTS. Since the matches have equal scores, a sub-optimal choice can be made here, leading to removed and added initial state arrows (Figure 5.2c). gLTSdiff features configurable scorers that allow taking into account extra information while computing scores, to influence subsequent state matching. For this example, considering initial state information ensures that we get the least number of differences (Figure 5.2d).



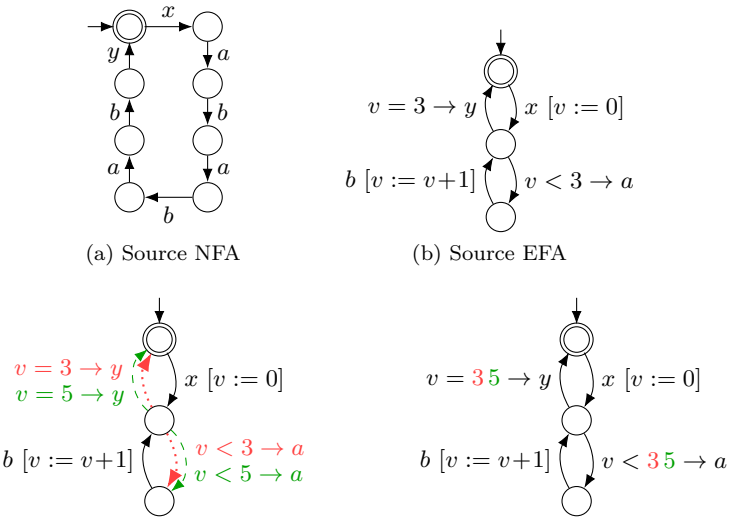
(a) Source LTS (b) Target LTS (c) Diff LTS (d) Improved Diff LTS

Figure 5.2: Example where considering initial states leads to fewer differences.

3) Accepting States: Using LTSDiff we have to ignore the extra state acceptance information of NFAs. gLTSdiff features configurable matchers. These can for instance be configured to allow two states to be matched only if they agree on acceptance. While scoring can be configured to influence matches, matchers can be configured to prevent certain states from being matched. Similar to adapting scoring for initial states, configuring matchers to account for state acceptance may lead to fewer differences in comparison results.

4) Repetitions/EFAs: In our work we regularly encounter NFAs that repeat the same behavior (Figure 5.3a, *ab* is repeated three times), for instance when polling is involved. An NFA can be made more compact by detecting such repetitions [98], and adapting it to an Extended Finite Automaton (EFA) with bounded (possibly nested) loops (Figure 5.3b). This makes its representation smaller, especially if sequences are longer or repeated many times. The EFA guards and updates can be encoded in the transition labels to represent the EFA as an NFA, allowing NFA comparison. However, this leads to differences that cover entire transitions, such as when the EFA is compared with a second input allowing the repetition 5 times (Figure 5.3c). By instead recursively comparing the structure of EFA transitions, more fine-grained differences can be obtained, revealing the repetition count as the only difference (Figure 5.3d). gLTSdiff supports such fine-grained comparison through configurable comparison of states and transitions.

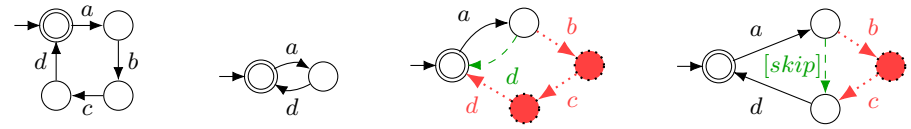
5) Skips: We also regularly encounter comparisons where one NFA skips a part of another (Figure 5.4), for example when behavior is added or removed in a new software version. Such comparisons result in Diff NFAs where either the transition before the skipped behavior is duplicated (fork pattern, Figure 5.2d, transition a from state '*'), or the one after it (join pattern, Figure 5.4c). In Figure 5.4c, d is present as both an added and a removed transition to the initial state, while that



(a) Source NFA (b) Source EFA

(c) Diff EFA (d) Improved Diff EFA

Figure 5.3: EFA-encoded repetitions, compared on full labels and sub-labels.



(a) Source NFA (b) Target NFA (c) Diff NFA (d) Improved Diff NFA

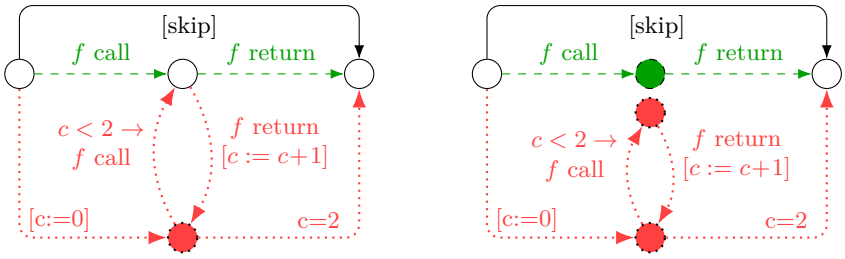
Figure 5.4: Example join skip pattern that is rewritten.

is not actually a difference. gLTSdiff features optional post-processing to rewrite such patterns, removing the duplication, leading to fewer differences (Figure 5.4d).

6) Tangles: We furthermore observe that compare results may contain states with only red and green incoming and outgoing transitions. Such 'tangles' are due to sub-optimal heuristic matching, where two entirely different states/paths got entangled. gLTSdiff features optional post-processing to untangle them, leading to more intuitive representations of the differences. Figure 5.5a shows an example, for paths where f is called one or two times. Untangling separates the paths where the function is called zero, one and two times (Figure 5.5b). However, in this example the paths are still quite related. If completely unrelated states and paths get entangled, the benefit of untangling is even more apparent.

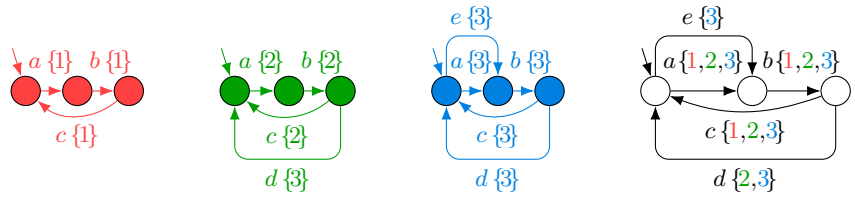
7) More Inputs: LTSDiff takes two LTSs as input, compares them, and produces a Diff LTS. Comparing n LTSs requires $\frac{n(n-1)}{2}$ comparisons, which does not scale. The many comparison results also make it difficult to see the bigger picture. gLTSdiff considers comparison a binary operator, allowing for example two Diff NFAs, one entirely red and the other entirely green, to be compared, resulting in a Diff NFA with their common parts merged (black), while their differences remain (in red and green). However, the transitions could for instance also be numbered (Figures 5.6a – 5.6c). Any number of input models can then be compared, by repeated execution of binary comparison on results of previous comparisons, producing a single difference model with all their differences (Figure 5.6d).

8) Performance: LTSDiff creates a system of linear equations, where for every state pair there is a variable, and the variables are related to each other, potentially leading to twice a quadratic factor. gLTSdiff has multiple scoring and matching algorithms, including faster and slower ones that may produce more or less differ-



(a) Tangled Diff EFA (b) Untangled Diff EFA

Figure 5.5: Example partial Diff EFA that is untangled.



(a) 1st input (b) 2nd input (c) 3rd input (d) Diff

Figure 5.6: Example comparison with three inputs (colors for visualization purposes only).

ences, allowing for an effort/quality trade-off. Dynamic scorers and matchers use slower algorithms for smaller models and faster ones for larger models.

9) Extensibility: The LTSDiff algorithm is implemented in the StateChum tool[1]. We experienced difficulties adapting and extending StateChum to address the described challenges. Our gLTSdiff library, implementing the gLTSdiff algorithm, is developed with extensibility for adding new algorithms and model representations as a key consideration.

5.3 The gLTSdiff framework

gLTSdiff generalizes and extends LTSDiff. Input LTSs are generalized to *gLTSs* with arbitrary state and transition properties, encoding a wide range of state machine models (Section 5.3.1). A *combinability* notion allows configuring which state and transition properties are *combinable* (mergeable). A *combiner* then allows to configure the result of *combining* (merging) two such properties (Section 5.3.2). Using combiners, gLTSdiff first computes a state *matching* with a state *matcher* (Section 5.3.3), for example using generalized versions of LTSDiff's scoring and matching (Section 5.3.4). Based on a matching, the *merger* then *merges* the input gLTSs into a single gLTS, combining properties of matched states and transitions as configured (Section 5.3.5). gLTSdiff is then a binary operation over gLTSs that performs matching and merging (Section 5.3.6). Finally, undesired patterns may be *rewritten* (Section 5.3.7).

5.3.1 Generalized labeled transition systems

We define a *generalized Labeled Transition System* (gLTS), an LTS with arbitrary state properties:

Definition 5.3 (Generalized Labeled Transition System)**.** *A gLTS* $L = (S, \mathbb{S}, P, \mathbb{T}, T)$ *is a 5-tuple with:* S *a finite set of states,* $\mathbb{S}$ *a set of state properties,* $P : S \to \mathbb{S}$ *a function that assigns properties to states,* $\mathbb{T}$ *a set of transition properties, and* $T \subseteq S \times \mathbb{T} \times S$ *a finite set of transitions.*

[1] See `https://github.com/kirilluk/statechum`.

Our definition resembles definitions from the literature, with subtle non-fundamental differences [33, 69, 100]. As before, subscripts are used to distinguish multiple gLTSs, and superscripts to refer to elements of gLTSs. We define, for a transition $t \in L^T$, t^{src} and t^{tgt} as for LTSs, renaming t^{lbl} to t^{prp}.

gLTSs allow encoding many well-known graph and state machine models. An LTS $L = (S, \Sigma, \Delta, I)$ can be encoded as a gLTS $L' = (S, \mathbb{B}, \lambda s.s \in I, \Sigma, \Delta)$, with state properties being Booleans from $\mathbb{B}$ expressing whether a state is initial. Similarly, an NFA can be encoded as a gLTS:

Definition 5.4 (gNFA)**.** *A gNFA is an NFA $A = (S, \Sigma, \Delta, I, F)$ encoded as a gLTS $A' = (S, \mathbb{B}\times\mathbb{B}, \lambda s.(s \in I, s \in F), \Sigma, \Delta)$, with states being initial (1^{st} $\mathbb{B}$) and/or final (2^{nd} $\mathbb{B}$).*

For example, for the gNFA of Figure 5.4b, we could define:

- $S = \{s_1, s_2\}$
- $P(s_1) = (\top, \top)$
- $P(s_2) = (\bot, \bot)$
- $T = \{(s_1, a, s_2), \quad (s_2, d, s_1)\}$

Extending gNFAs, gDiffNFAs encode Diff NFAs as gLTSs:

Definition 5.5 (gDiffNFA)**.** *A gDiffNFA is a gLTS A, with: $A^{\mathbb{S}} = \mathbb{B} \times \mathbb{B} \times D \times (D \cup \{\bot\})$ a set of state properties (initial, final, state difference kind, state initial difference kind or $\bot$ if state is not initial), $A^{\mathbb{T}} = \Sigma \times D$ a set of transition properties (symbol, symbol/transition difference kind), and $D = \{-, +, =\}$ a set of difference kinds (removed, added, unchanged).*

Both state and transition properties get additional difference kinds. For example, for the gDiffNFA of Figure 5.2c, we could define:

- $S = \{s_1, s_2, s_3\}$
- $P(s_1) = (\top, \bot, -, -)$
- $P(s_2) = (\bot, \bot, =, \bot)$
- $P(s_3) = (\top, \bot, =, +)$
- $T = \{(s_1, (c, -), s_2), \quad (s_2, (a, =), s_3), \quad (s_3, (c, +), s_2), \quad (s_3, (b, -), s_1)\}$

gNFAs are also extended to gEFAs, with variables, guards and updates (for this chapter kept simple, with only integers):

Definition 5.6 (gEFA)**.** *A gEFA is a gLTS A, with: state properties $A^{\mathbb{S}} = \mathbb{B} \times \mathbb{B}$ (initial, final), transition properties $A^{\mathbb{T}} = \mathcal{P}(E) \times \Sigma \times \mathcal{P}(U)$ (guards, symbol, updates), expressions $E = V \cup \mathbb{N} \cup (E \times O \times E)$ (variables, numbers, binary expressions), finite variable set V (distinct from $\mathbb{N}$), binary operators $O = \{\leq, <, =, \neq, >, \geq, +, -\}$, and updates $U = V \times E$.*

For example, for the gEFA of Figure 5.3b, we could define:

- $S = \{s_1, s_2, s_3\}$

- $P(s_1) = (\top, \top)$
- $P(s_2) = (\bot, \bot)$
- $P(s_3) = (\bot, \bot)$
- $T = \{(s_1, (\emptyset, x, \{(v, 0)\}), s_2), \quad (s_2, (\{(v, <, 3)\}, a, \emptyset), s_3), \quad (s_3, (\emptyset, b, \{(v, (v, +, 1))\}), s_2), \quad (s_2, (\{(v, =, 3)\}, y, \emptyset), s_1)\}$

Type safety is not essential for structural comparison. In figures, $(e_1, o, e_2) \in E \times O \times E$ is shown as '$e_1\, o\, e_2$', and $(v, e) \in U$ as '$v := e$'.

gDiffNFAs and gEFAs can be combined to form gDiffEFAs:

Definition 5.7 (gDiffEFA)**.** *A gDiffEFA is a gLTS A, with: $A^{\mathbb{S}} = \mathbb{B} \times \mathbb{B} \times D \times (D \cup \{\bot\})$, $A^{\mathbb{T}} = \mathcal{P}(E) \times (\Sigma \times D) \times \mathcal{P}(U)$, $E = (V \times D) \cup (\mathbb{N} \times D) \cup (E \times (O \times D) \times E) \cup \mathcal{P}(E)$, and $U = (V \times D) \times E$.*

In the structure of $\mathbb{T}$, all leaf elements (transition symbols, variables, numbers, operators and updated variables) get an extra D. Adding $\mathcal{P}(E)$ to E allows both added and removed sub-expressions at the same place within a larger expression. For example, for the gDiffEFA of Figure 5.3d, we could define:

- $S = \{s_1, s_2, s_3\}$
- $P(s_1) = (\top, \top, =, =)$
- $P(s_2) = (\bot, \bot, =, \bot)$
- $P(s_3) = (\bot, \bot, =, \bot)$
- $T = \{(s_1, (\emptyset, (x, =), \{((v, =), (0, =))\}), s_2), \quad (s_2, (\{((v, =), (<, =), \{(3, -), (5, +)\})\}, (a, =), \emptyset), s_3), \quad (s_3, (\emptyset, (b, =), \{((v, =), ((v, =), (+, =), (1, =)))\}), s_2), (s_2, (\{((v, =), (=, =), \{(3, -), (5, +)\})\}, (y, =), \emptyset), s_1)\}$

We further define version-annotated LTSs:

Definition 5.8 (gVLTS)**.** *A gVLTS is a gLTS A with: $A^{\mathbb{S}} = \mathbb{B}$ (initial), and $A^{\mathbb{T}} = \Sigma \times \mathcal{P}(\mathbb{N})$ (symbol, version numbers).*

For example, for the gVLTS of Figure 5.6a, we could define:

- $S = \{s_1, s_2, s_3\}$
- $P(s_1) = \top$
- $P(s_2) = \bot$
- $P(s_3) = \bot$
- $T = \{(s_1, (a, \{1\}), s_2), \quad (s_2, (b, \{1\}), s_3), \quad (s_3, (c, \{1\}), s_1)\}$

5.3.2 Combining state and transition properties

Now that we have generalized LTSs to gLTSs, with arbitrary state and transition properties, the notions of matching states and transitions need to be generalized as well. For instance, we may want to only allow merging NFA states that agree on state acceptance, and to define merging of EFA transitions as a recursive operation on their structure, highlighting differences on their leafs (see Figure 5.3). In general, we want to be able to configure when state and transition properties may be merged (are *combinable*), and if so, what the result is when merging them (their *combination*):

Definition 5.9 (Combiners and combinability)**.** *A* combiner $c : X \times X \rightharpoonup X$ *is a partial binary function over* X *that* combines *two properties to their* combination*. A combiner* c *induces a* combinability *relation* $\bowtie_c$*, defined as* $x_1 \bowtie_c x_2 \Leftrightarrow (x_1, x_2) \in \mathsf{dom}(c)$*. They must together satisfy the following conditions:*

1. $\bowtie_c$ *is an equivalence relation, i.e., it is 1.1) reflexive, 1.2) symmetric and 1.3) transitive,*
2. c *preserves* $\bowtie_c$*:* $x_1 \bowtie_c x_2 \implies x_1 \bowtie_c c(x_1, x_2)$*,*
3. c *is associative:* $x_1 \bowtie_c x_2 \ \wedge \ x_2 \bowtie_c x_3 \implies c(x_1, c(x_2, x_3)) = c(c(x_1, x_2), x_3)$*, and*
4. c *is commutative:* $x_1 \bowtie_c x_2 \implies c(x_1, x_2) = c(x_2, x_1)$*.*

The equivalence classes of a combiner under combinability are equivalent to a disjoint union of abelian semigroups. The four conditions ensure that any number of combinable properties may be combined, in any order. That is, conditions *1.2* and *1.3* ensure that properties may be combined in any order, condition *2* ensures that combinability is not lost along the way, and conditions *3* and *4* ensure that their combination is then always the same, regardless of the order. Condition *1.1* ensures that properties may be combined with themselves, for instance allowing identical transition labels of multiple gEFAs to be combined.

For convenience, we extend c from a binary to an n-ary operation, by defining $c(x_1, x_2, ..., x_n) = c(c(c(x_1, x_2), ...), x_n)$. Such a combination is only allowed if the properties $x_1, x_2, ..., x_n$ are combinable through the associated notion of combinability $\bowtie_c$, which we define for $n > 0$ as:

$$\bowtie_c(x_1, x_2, ..., x_n) = \begin{cases} \top & \text{if } n = 1 \\ \bowtie_c(x_1, x_2) & \text{if } n = 2 \\ \bowtie_c(c(x_1, x_2, ..., x_{n-1}), x_n) & \text{otherwise} \end{cases}$$

That is, a single property doesn't need to be combined, and is thus always combinable ($\top$). This matches with condition *1.1* that $\bowtie_c$ is reflexive, and properties can thus always be combined with themselves. For two properties, we fall back to Definition 5.9. For more than two properties, we define combinability recursively.

For a combiner c there is always an associated notion of combinability $\bowtie_c$, and vice versa, even if not explicitly mentioned. Where possible we name combiners c and combinability $\bowtie$ based on what they combine or how they combine it, writing for instance $c_{\mathbb{S}}$ for a combiner over state properties from $\mathbb{S}$, and $\bowtie_{\mathbb{S}}$ rather than $\bowtie_{c_{\mathbb{S}}}$, leaving combiner c implicit.

This theoretical framework allows defining composite combiners over the structure of state and transition properties:

- *Pairs:* For any two combiners c_X over some set X and c_Y over some sets Y, combiner $c_\times$ combines pairs from $X \times Y$ by combining their elements. Formally, for any $x_1, x_2 \in X$ and $y_1, y_2 \in Y$, we define $(x_1, y_1) \bowtie_\times (x_2, y_2) \Leftrightarrow x_1 \bowtie_X x_2 \wedge y_1 \bowtie_Y y_2$. For combinable pairs, $c_\times((x_1, y_1), (x_2, y_2)) = (c_X(x_1, x_2), c_Y(y_1, y_2))$. This also generalizes to n-tuples.

- *Sets:* For any combiner c_X over some set X, combiner $c_{\mathcal{P}}$ combines sets from powerset $\mathcal{P}(X)$ by combining the equivalence classes of their elements. Formally, for any sets $X_1, X_2 \in \mathcal{P}(X)$, $X_1 \bowtie_{\mathcal{P}} X_2 = \top$, and $c_{\mathcal{P}}(X_1, X_2) = \{c_X(Y) \,|\, Y \in (X_1 \cup X_2)/\bowtie_X\}$. Here, Y is a single equivalence class of combinable elements, and c_X combines the combinable elements to a single element using the n-ary extension of c_X.
- *Optionals:* For any combiner c_X over some set X, combiner $c_\perp$ combines optional values from set $X \cup \{\perp\}$, with $\perp \notin X$. Formally, if $\forall_{x,x' \in X}\, x \bowtie_X x' = \top$, then for any $x_1, x_2 \in X \cup \{\perp\}$, $x_1 \bowtie_\perp x_2 = \top$ and $c_\perp(\perp, \perp) = \perp$, $c_\perp(x_1, \perp) = x_1$, $c_\perp(\perp, x_2) = x_2$, and otherwise $c_\perp(x_1, x_2) = c_X(x_1, x_2)$. Assumption $\forall_{x,x' \in X}\, x \bowtie_X x' = \top$ ensures that $\bowtie_\perp$ is transitive.

Basic combiners for leaf types can also be defined, such as:

- *Equality:* Equality combiner $c_=$ over any set X only allows combining equal properties, and leaves them unchanged. Formally, for any $x_1, x_2 \in X$, $x_1 \bowtie_= x_2 \Leftrightarrow x_1 = x_2$. For combinable properties, $c_=(x_1, x_2) = x_1$.
- *Disjunction of Booleans:* Combiner $c_\vee$ allows combining any two Booleans, producing their disjunction. Formally, for any $b_1, b_2 \in \mathbb{B}$, $b_1 \bowtie_\vee b_2 \Leftrightarrow \top$ and $c_\vee(b_1, b_2) = b_1 \vee b_2$.
- *Difference kinds:* For any $d_1, d_2 \in D$, $d_1 \bowtie_D d_2 \Leftrightarrow \top$, and $c_D(-, -) = \text{`}-\text{'}$, $c_D(+, +) = \text{`}+\text{'}$ and else $c_D(d_1, d_2) = \text{`}=\text{'}$.

Similarly, composite and basic combiners may also be defined for other types. We write composite combiners in terms of the combiners that they rely upon, e.g., $c_\times[c_=, c_D]$ for a pair combiner that uses $c_=$ for left and c_D for right elements of pairs, and similarly $\bowtie_\times[\bowtie_=, \bowtie_D]$. This way, combiners and combinability may be reused and composed in different ways.

We now present the combiners for the representations used in this chapter.

gDiffNFAs: For gDiffNFAs, we use the following combiners:

$$\begin{aligned} c_{\mathbb{S}} &= c_\times[c_\vee, c_=, c_D, c_\perp[c_D]] \\ c_{\mathbb{T}} &= c_\times[c_=, c_D] \end{aligned}$$

Initial states are later taken into account for scoring, while merging states that disagree on acceptance is explicitly prevented here. The different handling of initial and acceptance information stems from practical experience.

gDiffEFAs: For gDiffEFAs, we use the following combiners:

$$\begin{aligned} c_{\mathbb{S}} &= c_\times[c_\vee, c_=, c_D, c_\perp[c_D]] \\ c_{\mathbb{T}} &= c_\times[c_{\mathcal{P}}[c_E], c_\times[c_=, c_D], c_{\mathcal{P}}[c_U]] \\ c_U &= c_\times[c_\times[c_=, c_D], c_E] \\ \bowtie_E(e_1, e_2) &= \text{true} \end{aligned}$$

$$c_E(e_1, e_2) = \begin{cases} c_\times[c_=, c_D](e_1, e_2) \\ \quad \text{if } e_1, e_2 \in V \times D \wedge e_1 \bowtie_\times[\bowtie_=, \bowtie_D]\ e_2 \\ c_\times[c_=, c_D](e_1, e_2) \\ \quad \text{if } e_1, e_2 \in \mathbb{N} \times D \wedge e_1 \bowtie_\times[\bowtie_=, \bowtie_D]\ e_2 \\ c_\times[c_E, c_\times[c_=, c_D], c_E](e_1, e_2) \\ \quad \text{if } e_1, e_2 \in E \times (O \times D) \times E \\ \quad\quad \wedge e_1 \bowtie_\times[\bowtie_E, \bowtie_\times[\bowtie_=, \bowtie_D], \bowtie_E]\ e_2 \\ c_\mathcal{P}[c_E](e_1, e_2) \text{ if } e_1, e_2 \in \mathcal{P}(E) \wedge e_1 \bowtie_\mathcal{P}[\bowtie_E]\ e_2 \\ c_\mathcal{P}[c_E](e_1, \{e_2\}) \\ \quad \text{if } e_1 \in \mathcal{P}(E), e_2 \notin \mathcal{P}(E) \wedge e_1 \bowtie_\mathcal{P}[\bowtie_E]\ \{e_2\} \\ c_\mathcal{P}[c_E](\{e_1\}, e_2) \\ \quad \text{if } e_1 \notin \mathcal{P}(E), e_2 \in \mathcal{P}(E) \wedge \{e_1\} \bowtie_\mathcal{P}[\bowtie_E]\ e_2 \\ \{e_1, e_2\} \text{ otherwise} \end{cases}$$

The extensions follow directly from the structure of expressions and updates. If expression operators don't match, the entire binary expression is not combined recursively. An interested reader could adapt this to recursively still consider the operands.

gVLTSs: For gVLTSs, we use the following combiners:

$$c_\mathbb{S} = c_=$$
$$c_\mathbb{T} = c_\times[c_=, c_\mathcal{P}[c_=]]$$

gVLTSs encode no state difference information, so only states with matching initial state properties are merged.

5.3.3 Matching gLTSs

We define a valid *matching* of states from two input gLTSs:

Definition 5.10 (Matching)**.** *For any two gLTSs L_1 and L_2 over the same set of state properties (i.e., $L_1^\mathbb{S} = L_2^\mathbb{S}$), and $c_\mathbb{S}$ a state property combiner, any injective partial function $m : L_1^S \rightharpoonup L_2^S$ is a* matching *for L_1 and L_2 with respect to $c_\mathbb{S}$ if $s \in \mathsf{dom}(m) \implies L_1^P(s) \bowtie_\mathbb{S} L_2^P(m(s))$.*

Matchings are finite as gLTSs have finite sets of states. Any matching m being injective means that m is a one-to-one mapping, i.e., all states of L_1 and L_2 are mapped to at most one state of the other. This makes m a matching (or assignment) in the standard graph-theoretical sense.

gLTSdiff uses a matcher to obtain a state matching, and then merges each pair of matched states into a single state, thereby combining their state properties. We therefore impose the additional constraint on matchings that the properties of all matched states must be combinable. This may in turn be exploited by implementations of matchers, to reduce the search space of potential matches, improving performance.

The following definition gives a *specification* for matchers:

Definition 5.11 (Matcher)**.** *A* matcher *$matcher(L_1, L_2, c_\mathbb{S}, c_\mathbb{T}) : L_1^S \rightharpoonup L_2^S$ is an operation that computes a matching for its two input gLTSs L_1 and L_2 over the same sets of state and transition properties (i.e., $L_1^\mathbb{S} = L_2^\mathbb{S}$ and $L_1^\mathbb{T} = L_2^\mathbb{T}$), and state and transition property combiners $c_\mathbb{S}$ and $c_\mathbb{T}$, respectively.*

Many possible implementations of matchers exist that satisfy this specification, including heuristic-based ones like LTSDiff, and optimization-based ones like the well-known Kuhn-Munkres algorithm (also called the Hungarian method).

5.3.4 Generalizing scoring and matching

The definitions of global scoring and matching as defined in LTSDiff require adaptations in gLTSdiff, to support gLTSs, combiners and combinability. For $s_1 \in L_1^S$ and $s_2 \in L_2^S$, we generalize common successors *succ* from Equation 5.1 (see Section 5.1.4) to combinable outgoing transitions $succ_c$. Furthermore, we define $succ_{un}(s_1, s_2)$, the outgoing transitions of s_1 whose properties can not be combined with those of any outgoing transition of s_2 (and similarly the other way around):

$$\begin{aligned} succ_c(s_1, s_2) = \{(t_1, t_2) \mid t_1 \in L_1^T \wedge t_2 \in L_2^T \wedge \\ t_1^{src} = s_1 \wedge t_2^{src} = s_2 \wedge t_1^{prp} \bowtie_{\mathbb{T}} t_2^{prp}\} \\ succ_{un}(s_1, s_2) = \{t_1 \mid t_1 \in L_1^T \wedge t_1^{src} = s_1 \wedge \\ \nexists_{t_2 \in L_2^T} \ (t_2^{src} = s_2 \wedge t_1^{prp} \bowtie_{\mathbb{T}} t_2^{prp})\} \end{aligned}$$

With this, we generalize global successor scoring from Equation 5.2:

$$sc_{succ}^{glob}(s_1, s_2) = \frac{1}{2} \frac{\sum_{(t_1,t_2) \in succ_c(s_1,s_2)} \left(1 + k \cdot sc_{succ}^{glob}(t_1^{tgt}, t_2^{tgt})\right)}{|succ_{un}(s_1, s_2)| + |succ_{un}(s_2, s_1)| + |succ_c(s_1, s_2)|}$$

Any two states that do not have combinable state properties get score -1 instead. We then get $|sc_{succ}^{glob}| \leq 1$ rather than $sc_{succ}^{glob} \leq 1$, as defined by Walkinshaw and Bogdanov [126], who similarly allow negative scores. We define sc^{glob} to be $-\infty$ if either sc_{pred}^{glob} or sc_{succ}^{glob} is negative, and otherwise use their average as in Equation 5.3. Scores are thus extended with $-\infty$, indicating to score-based matchers that the state pair must never be matched. They should thus, for example, not be considered for landmarks and fallback landmarks. In general, both score-based and non-score-based matchers never match states with non-combinable state properties.

Furthermore, to allow taking into account state property information that is not captured in combiners and combinability, the scoring fractions may optionally be customized. For gLTSs with initial states, $sc_{pred}^{glob}(s_1, s_2)$ is adapted, by incrementing the dividend by one if both states are initial, and the divisor by one if either of the states is initial. The fallback landmarks can also be customized. The fallback to initial states is omitted for gLTSs without initial states.

5.3.5 Merging gLTSs

gLTSdiff uses a matching as a basis for merging two input gLTSs into a single merged gLTS. Matched states and transitions are merged into single states and transitions, combining their properties using combiners:

Definition 5.12 (Merger)**.** *The* merger *is an operation that computes the merged gLTS L for two input gLTSs L_1 and L_2 over the same types of state properties*

$\mathbb{S}$ *and transition properties* $\mathbb{T}$*, but with disjoint states* ($L_1^S \cap L_2^S = \emptyset$)*, given a state combiner* $c_{\mathbb{S}}$*, a transition combiner* $c_{\mathbb{T}}$*, and* m *a matching over* L_1 *and* L_2*. Formally,* $merger(L_1, L_2, c_{\mathbb{S}}, c_{\mathbb{T}}, m) = L$*, with:*

- $f_1 : L_1^S \to L^S$*, with* $f_1(s) = s$
- $f_2 : L_2^S \to L^S$*, with* $f_2(s) = \begin{cases} s & \text{if } s \in L_2^S \setminus \mathsf{rng}(m) \\ m^{-1}(s) & \text{otherwise} \end{cases}$
- $L^S = \{f_1(s) \mid s \in L_1^S\} \cup \{f_2(s) \mid s \in L_2^S\}$
- $L^P(s) = \begin{cases} L_1^P(s) & \text{if } s \in L_1^S \setminus \mathsf{dom}(m) \\ L_2^P(s) & \text{if } s \in L_2^S \setminus \mathsf{rng}(m) \\ c_{\mathbb{S}}(L_1^P(s), L_2^P(m(s))) & \text{otherwise} \end{cases}$
- $T_1 = \{(f_1(s_1), t, f_1(s_2)) \mid (s_1, t, s_2) \in L_1^T\}$
- $T_2 = \{(f_2(s_1), t, f_2(s_2)) \mid (s_1, t, s_2) \in L_2^T\}$
- $L^T = c_{\mathcal{P}}[c_{\times}[c_{=}, c_{\mathbb{T}}, c_{=}]](T_1 \cup T_2)$

L^S consists of the states of the two input gLTSs, mapped via functions f_1 and f_2. Unmatched states are used as is, while matched states from the first input gLTS are reused. Without loss of generality, we require $L_1^S \cap L_2^S = \emptyset$ as precondition to prevent state overlap issues. State properties from unmatched states are preserved. For matched states, the properties are combined using $c_{\mathbb{S}}$. Sets T_1 and T_2 contain the transitions of the two inputs gLTSs, with their source and target states mapped to states in L^S. Transition set L^T then contains the combined transitions from T_1 and T_2. Combiner $c_{\times}$ combines transition triples, where source and target states must match ($\bowtie_{=}$) and their transition properties must be combinable ($\bowtie_{\mathbb{T}}$) for the transitions to be combinable, which leads to their transition properties being combined ($c_{\mathbb{T}}$). Combiner $c_{\mathcal{P}}$ merges sets of transitions, combining all combinable transitions.

5.3.6 Generalized structural comparison

Finally, the gLTSdiff algorithm is defined:

Definition 5.13 (gLTSdiff)**.** *Generalized structural comparison is a binary operator that given any matcher and merger over gLTSs with the same types of state properties* $\mathbb{S}$ *and transition properties* $\mathbb{T}$*, constructs a single merged gLTS also over* $\mathbb{S}$ *and* $\mathbb{T}$*. Formally:*

$$gLTSdiff_{c_{\mathbb{S}}, c_{\mathbb{T}}}(L_1, L_2) = merger(L_1, L_2, c_{\mathbb{S}}, c_{\mathbb{T}}, matcher(L_1, L_2, c_{\mathbb{S}}, c_{\mathbb{T}}))$$

We omit subscripts $c_{\mathbb{S}}$ and $c_{\mathbb{T}}$ if they are clear from context. We use the same types of state and transition properties for the inputs as well as the output, as mentioned in challenge 7, and as is also be clear from the definition of the merger. For instance, two gDiffNFAs, one entirely in red and the other entirely in green, can be compared, resulting in a gDiffNFA with their common parts merged (in black), while their differences remain (in red and green). The inputs and output

are then all gDiffNFAs, rather than the inputs being gNFAs and only the output being a gDiffNFA.

This allows us to extend the algorithm to compare any number of inputs, by repeated application of *gLTSdiff* on results of previous comparisons:

$$gLTSdiff(L_1, L_2, ..., L_n) = gLTSdiff(gLTSdiff(gLTSdiff(L_1, L_2), ...), L_n)$$

It does not make sense to compare and merge more than two gDiffNFAs. Once a third input comes into play, two input colors (red and green) is not sufficient, and a third color for the third input would be required. It is thus essential that a gLTS representation is chosen for which it makes sense to compare and merge more than two inputs, such as one where the identities of the various inputs can still be recognized. The transitions could for instance be numbered, as in Figures 5.6a – 5.6c. Any number of input models can then be compared, by repeated application of *gLTSdiff* on results of previous comparisons:

$$\begin{aligned} L_{(1,2)} &= gLTSdiff(L_1, L_2) \\ L_{(1,2,3)} &= gLTSdiff(L_{(1,2)}, L_3) \\ L_{(1,2,3,4)} &= gLTSdiff(L_{(1,2,3)}, L_4) \\ L_{(1,2,3,4,5)} &= \dots \end{aligned}$$

That is, the first two inputs are compared and merged. Then, the third input is compared to and merged with the result of merging the first two inputs. The fourth input is compared to and merged with the result of merging the first three inputs, and so on. This way, a single single difference model with all their differences is produced, as in Figure 5.6d.

5.3.7 Rewriting undesired difference patterns

Ideally, a result of structural comparison has as few differences as possible, that are immediately apparent. Even with an optimization algorithm producing optimal matches, there may be 'undesired' differences. Taking a practical approach, we rewrite some of the undesired patterns that we encountered in our work: skips and tangles. Rewriting may optionally be performed as post-processing, after applying gLTSdiff. In this chapter, we explain only tangle rewriting in more detail.

Tangles, poorly matched (and thus merged) states with incoming and outgoing transitions of both inputs, where none of those transitions got merged (see Figure 5.5), are untangled:

Definition 5.14 (Tangle Rewriting)**.** *For a gDiffNFA L, any state $s \in L^S$ is a tangle if s is an unchanged state: $L^P(s)[3] = \text{`}=\text{'}$, with only added/removed incoming and outgoing transitions: $\{d \mid \exists_{t \in L^T}\ s \in \{t^{src}, t^{dst}\} \wedge t^{prp} = (\sigma, d)\} = \{+, -\}$. The tangle state can be rewritten by transforming L to L_r, with:*

- $L_r^S = (L^S \setminus \{s\}) \cup \{s_+, s_-\}$
- $i_+ = L^P(s)[1] \wedge L^P(s)[4] \neq \text{`}-\text{'}$
- $i_- = L^P(s)[1] \wedge L^P(s)[4] \neq \text{`}+\text{'}$
- $L_r^P(s_+) = (i_+, L^P(s)[2], +,$ *if* i_+ *then* `+' *otherwise* $\perp)$

- $L_r^P(s_-) = (i_-, L^P(s)[2], -,$ *if* i_- *then* '$-$' *otherwise* $\bot)$
- $L_r^T = \{(m(t^{src}), t^{prp}, m(t^{tgt})) \mid t \in L^T \wedge t^{prp} = (\sigma, d)\}$
  with for $s' \in \{t^{src}, t^{tgt}\}$, $m(s') = \begin{cases} s' & \text{if } s' \neq s \\ s_+ & \text{if } s' = s \wedge d = \text{'}+\text{'} \\ s_- & \text{if } s' = s \wedge d = \text{'}-\text{'} \end{cases}$

As an example, consider again Figure 5.5. In Figure 5.5a, there are three unchanged/black states. The middle one is a tangle state (s in Definition 5.14), with both removed/red incoming and outgoing edges, and added/green incoming and outgoing edges, but no unchanged/black ones. By rewriting, s is split into two new states not yet in L^S, with appropriate state properties, as shown in Figure 5.5b:

- State s_- is a removed/red state and s_+ is an added/green state.
- State s_+ is initial (as captured in i_+) if two conditions hold. First, s must be initial, per condition $L^P(s)[1]$. Second, s must not only be initial for removed (red/removed initial arrow). It must thus be initial for added (added/green initial arrow) or initial for both (unchanged/black initial arrow). This is captured by condition $L^P(s)[4] \neq$ '$-$'. And similarly i_- is defined for s_-. In the example, s is not initial, and thus neither of the two new states is initial.
- The final state property is inherited from s, using property $L^P(s)[2]$. In the example, s is not final, and thus the two new states are not final.
- The state initial difference kind for a new state that is initial, is equal to the new state's state difference kind, and it is $\bot$ if the new state is not initial. In the example, s is not initial and thus both new states get $\bot$ as state initial difference kind.

The incoming and outgoing transitions of s are moved to either s_+ or s_-. For a transition $t \in L^T$, the transition property t^{prp} is kept. The source and target states are mapped via m. If a source state s' of t is state s, then m maps it to s_+ if the transition is an added transition ($d =$ '$+$'), or to s_- if it is a removed transition ($d =$ '$-$'). And similarly for any target state s' of t that is state s. So, all removed/red incoming and outgoing transitions of s are moved to s_-, and all added/green incoming and outgoing transitions of s are moved to s_+.

Untangling models other than gDiffNFAs is future work.

5.4 The gLTSdiff open-source library

We have implemented our gLTSdiff algorithm in an open-source Java library. It is freely available under the MIT license on GitHub[2]. The library has been developed with extensibility, for adding new model representations and new algorithms, as a key consideration.

The `glts` package contains model representations, such as gLTSs, gNFAs and gDiffNFAs. `combiners` contains reusable combiners, including leaf combiners such as $c_=$ and c_D, and composite combiners such as $c_\times$ and $c_\mathcal{P}$. `matchers` contains matchers, such as an LTSDiff landmark-based matcher, a Kuhn-Munkres

[2]See `https://github.com/TNO/gLTSdiff`.

score-minimizing matcher that computes a guaranteed maximal matching (as few red/green states in the result as possible), and a dynamic matcher. `scorers` contains scorers for scoring-based matchers, such as local and global scorers as defined in LTSDiff, and a dynamic scorer. `mergers` contains the merger. `rewriters` contains rewriters, including fork/join skip and tangle rewriters.

Each package contains an interface or base class that allows multiple implementations for extensibility, and has one or more implementations already available. The various matchers allow for making a trade-off between computational effort and quality of the results, as a fast heuristics-based matching algorithm may produce sub-optimal results (such as the LTSDiff landmark-based matcher), while a computationally-intensive optimization algorithm may produce (more) optimal results (such as the Kuhn-Munkres score-minimizing matcher). The dynamic matcher automatically chooses a more computationally-intensive algorithm for smaller inputs and a faster one for larger inputs. Similarly, gLTSdiff features multiple scorers and a dynamic scorer. The local scorer has been extended to allow iterative refinement with a configurable bound, to take into account more context than a pure local scorer, while still performing less work than a global scorer. Some further details on several of the scorers and matchers are provided as part of the evaluation in Section 5.5.5.

The library has optimizations to improve performance, such as the global scorer precomputing as much as possible, to reduce the size and complexity of the system of linear equations. The library also comes with documentation on how to use and extend it.

5.5 Evaluation

We first show the practical value of gLTSdiff by applying it in industry (Sections 5.5.1 and 5.5.2). We then show that gLTSdiff handles large numbers of input models (Section 5.5.3), addressing challenge 7 and partially challenge 8. We further show that that gLTSdiff reduces the number of differences in comparison results (Section 5.5.4), addressing challenges 2 – 6. Finally, we show that gLTSdiff supports the effort/quality trade-off (Section 5.5.5), addressing challenge 8. This chapter's artifact contains all input models and code to reproduce the experiments [57].

5.5.1 Wafer exposure regression case study

To show the practical value of our work, we apply gLTSdiff to a case study [102] at ASML, a leading company in developing lithography systems. We thereby expand upon Chapter 4, comparing the behavior of different software versions to find potential behavioral regressions. The case study involves a high-level wafer exposure controller. A system throughput regression was found during pre-delivery testing of a new software version. ASML experts diagnosed it in about two person-days, tracing it back to a small change that was originally considered harmless, and was made for a different machine type than for which the regression was found. Existing tests did not find this issue.

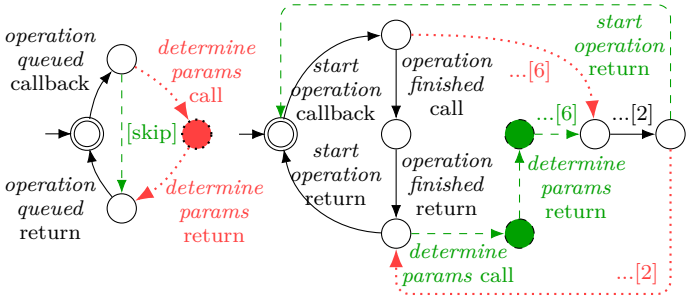


Figure 5.7: Wafer exposure case study: two anonymized compare results.

ASML executed a standard scenario, both for the software revision before and after the regression was introduced. In both cases, the resulting execution logs capture all communications in the system during execution, from which we inferred state machine models (see Chapter 3). Not knowing the cause of the regression, we applied gLTSdiff to compare the models, per function.

We looked into each function with behavioral changes, without having domain knowledge of this component. Figure 5.7 shows what we believed to be the root cause of the regression. In the *operation queued* callback (left in Figure 5.7), the *determine params* call is no longer executed. It has moved to the later *start operation* callback (right in Figure 5.7), causing the performance degradation. Some other functionality, abbreviated to '...[6]' in green, is now also executed there. The functionality that was originally here, abbreviated to '...[6]' and '...[2]' in red, has also been delayed, being moved to a third not shown callback. ASML experts confirm that we found the root cause, subsequent 'collateral' moves, and some remaining irrelevant as well as intended differences.

With our approach, we were able to find the differences in about two person-hours. Would it have been applied before the change was delivered to the integration team, this regression would not have happened. It could thus have been prevented, as was confirmed by the ASML component owner.

5.5.2 Wafer stage refactoring case study

To further show the practical value of our work, we apply gLTSdiff to a second ASML case study [102]. It involves code for a high-level wafer stage controller being split into two code bases for different machine types, one to be developed further, the other not. This redesign is considered particularly risky, as it involves a legacy component, there is time pressure, test machines are scarce or no longer available, and the team lacks knowledge of some old machine types. Using the same approach as for the first case study, models are obtained for executions from before and after the changes. After comparing them using gLTSdiff, an ASML engineer interpreted the results, while working on the refactoring. It took about half a person-day for each new code base, to analyze the results, discuss them in the redesign team, and to fix the identified regressions.

A prominent result is finding and fixing a serious regression, before the newly split code was delivered. The regression is shown in Figure 5.8. The *switch* function

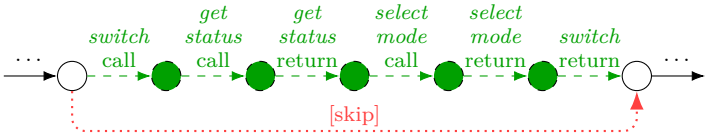


Figure 5.8: Wafer stage case study: anonymized partial compare result.

was not meant to be called here, for this machine type. The switch to a different control mode did not make the existing tests fail. It would have later shown up as a system throughput regression. Tracing it back to this particular call would have been "very hard", requiring significant diagnosis time, according to ASML engineers. Their very conservative estimate indicates that at least one person-day of effort is saved by preventing this regression from being delivered. Applying gLTSdiff also helped to increase confidence in the results of the risky redesign.

5.5.3 TLS implementations case study

To show how gLTSdiff can work with a large number of inputs, we apply it to merge Mealy machines representing the behavior of different server implementations for versions 1.0, 1.1 and 1.2 of the Transport Layer Security (TLS) protocol. We use the 332 Mbed TLS models and 264 OpenSSL models by Erwin Janssen [72], for a total of 596 input models, each with 6 to 14 states. Each model represents the behavior of a specific version of an implementation for a specific version of the TLS protocol, e.g., TLS 1.2 behavior of OpenSSL v1.1.0e.

To support comparing Mealy machines, we extend a gVLTS to a gVMM (Version-annotated Mealy Machine). We do this by splitting symbols Σ into Σ_i and Σ_o, such that $\mathbb{T} = \Sigma_i \times \Sigma_o \times \mathcal{P}(\mathbb{N})$, with $c_{\mathbb{T}} = c_{\times}[c_{=}, c_{=}, c_{\mathcal{P}}[c_{=}]]$. We feed all the models into gLTSdiff, to produce a single merged gVMM of all their behaviors. As all input models are complete Mealy machines, they have a transition for every input symbol in every state. When in a certain state the TLS protocol does not allow a certain command, such a command leads to a sink state. We remove the single sink state of the merged gVMM, which then has 13 states and 39 transitions left. By removing the sink state, we get some deadlock states, where no further commands are allowed by any version of the TLS protocol. Each transition is version-annotated with a set indicating which of the 596 models include that transition. As these sets can be large, and there are only 30 unique behaviors, we replace versions with equivalent behavior by a single unique number (see Table 5.1). The resulting gVMM is shown in Figure 5.9, where transitions are labeled with 'input / output(s) {behavior-number-ranges}'.

The merged gVMM was obtained in 30 seconds on a laptop with a 3 GHz Intel Core i7-1185G7 processor, using the LTSDiff landmarks-based matcher and optimized global scorer. Comparing instead each model pairwise to each other model leads to 354,620 comparisons, which takes about 14 minutes.

Nr	Impl.	TLS	Versions
1	Mbed TLS	1.0	1.0.0, 1.1.0, 1.1.1, 1.1.2, 1.1.3, 1.1.4, 1.1.5, 1.1.6, 1.1.7, 1.1.8
		1.1	1.0.0, 1.1.0, 1.1.1, 1.1.2, 1.1.3, 1.1.4, 1.1.5, 1.1.6, 1.1.7, 1.1.8
2	Mbed TLS	*	1.2.1, 1.2.2, 1.2.3, 1.2.4, 1.2.5, 1.2.6, 1.2.7, 1.2.8, 1.2.9, 1.2.10, 1.2.11, 1.3.6, 1.3.7, 1.3.8
3	Mbed TLS	*	1.3.0, 1.3.1, 1.3.2, 1.3.3, 1.3.4, 1.3.5
4	Mbed TLS	*	2.0.0, 2.1.0, 2.1.1, 2.1.2, 2.1.3, 2.1.4, 2.1.5, 2.1.6, 2.1.7, 2.1.8, 2.1.9, 2.1.10, 2.1.11, 2.1.12, 2.1.13, 2.1.14, 2.1.15, 2.1.16, 2.1.17, 2.1.18, 2.2.0, 2.2.1, 2.3.0, 2.4.0, 2.4.1, 2.4.2, 2.5.0
5	Mbed TLS	*	2.5.1, 2.6.0, 2.6.1, 2.7.0, 2.7.1, 2.7.2, 2.7.3, 2.7.4, 2.7.5, 2.7.6, 2.7.7, 2.7.8, 2.7.9, 2.7.10, 2.7.11, 2.7.12, 2.7.13, 2.7.14, 2.7.15, 2.7.16, 2.7.17, 2.8.0, 2.9.0, 2.10.0
6	Mbed TLS	*	2.11.0, 2.12.0, 2.13.0, 2.13.1, 2.14.0, 2.14.1, 2.15.0, 2.15.1, 2.16.0, 2.16.1, 2.16.2, 2.16.3, 2.16.4, 2.16.5, 2.16.6, 2.16.7, 2.16.8, 2.17.0, 2.18.0, 2.18.1, 2.19.0, 2.19.0d1, 2.19.0d2, 2.19.1, 2.20.0, 2.20.0d0, 2.20.0d1, 2.21.0, 2.22.0, 2.22.0d0, 2.23.0, 2.24.0, 3.0.0p1
7	OpenSSL	1.0	0.9.7, 0.9.7a, 0.9.7b, 0.9.7c, 0.9.7d
8	OpenSSL	1.0	0.9.7e, 0.9.7f, 0.9.7g, 0.9.7h, 0.9.7i, 0.9.7j, 0.9.7k, 0.9.7l, 0.9.7m, 0.9.8, 0.9.8a, 0.9.8b, 0.9.8c, 0.9.8d, 0.9.8e, 0.9.8f, 0.9.8g, 0.9.8h, 0.9.8i, 0.9.8j, 0.9.8k
9	OpenSSL	1.0	0.9.8l
10	OpenSSL	1.0	0.9.8m, 0.9.8n, 0.9.8o, 0.9.8p, 0.9.8q, 0.9.8r, 1.0.0, 1.0.0a, 1.0.0b, 1.0.0c, 1.0.0d, 1.0.0e
11	OpenSSL	1.0	0.9.8s, 0.9.8t
12	OpenSSL	1.0	0.9.8u, 0.9.8v, 0.9.8w, 0.9.8x
13	OpenSSL	1.0	0.9.8y
14	OpenSSL	1.0	0.9.8za
15	OpenSSL	1.0	0.9.8zb, 0.9.8zc, 0.9.8zd, 0.9.8ze, 0.9.8zf, 0.9.8zg, 0.9.8zh, 1.0.0n, 1.0.0o, 1.0.1i, 1.0.1j
		1.1	1.0.1i, 1.0.1j
		1.2	1.0.1i, 1.0.1j
16	OpenSSL	1.0	1.0.0f
17	OpenSSL	1.0	1.0.0g
18	OpenSSL	1.0	1.0.0h, 1.0.0i, 1.0.0j, 1.0.1, 1.0.1a, 1.0.1b, 1.0.1c
19	OpenSSL	1.0	1.0.0k, 1.0.0l, 1.0.1d
20	OpenSSL	1.0	1.0.0m
21	OpenSSL	1.0	1.0.0p, 1.0.0q, 1.0.0r, 1.0.0s, 1.0.0t
22	OpenSSL	1.1	1.0.1, 1.0.1a, 1.0.1b, 1.0.1c
		1.2	1.0.1, 1.0.1a, 1.0.1b, 1.0.1c
23	OpenSSL	1.1	1.0.1d
		1.2	1.0.1d
24	OpenSSL	*	1.0.1e, 1.0.1f, 1.0.1g
25	OpenSSL	*	1.0.1h
26	OpenSSL	*	1.0.1k, 1.0.1l, 1.0.1m, 1.0.1n, 1.0.1o, 1.0.1p, 1.0.1q, 1.0.1r, 1.0.1s, 1.0.1t, 1.0.1u
27	OpenSSL	*	1.0.2, 1.0.2a, 1.0.2b, 1.0.2c, 1.0.2d, 1.0.2e, 1.0.2f, 1.0.2g, 1.0.2h, 1.0.2i, 1.0.2j, 1.0.2k, 1.0.2l
28	OpenSSL	*	1.0.2m, 1.0.2n, 1.0.2o, 1.0.2p, 1.0.2q, 1.0.2r, 1.0.2s, 1.0.2t, 1.0.2u
29	OpenSSL	*	1.1.0, 1.1.0a, 1.1.0b, 1.1.0c, 1.1.0d, 1.1.0e, 1.1.0f, 1.1.0g, 1.1.0h, 1.1.0i, 1.1.0j, 1.1.0k, 1.1.0l
30	OpenSSL	*	1.1.1, 1.1.1a, 1.1.1b, 1.1.1c, 1.1.1d, 1.1.1e, 1.1.1f, 1.1.1g

Table 5.1: Unique TLS behavior numbers mapped to TLS server implementation versions.

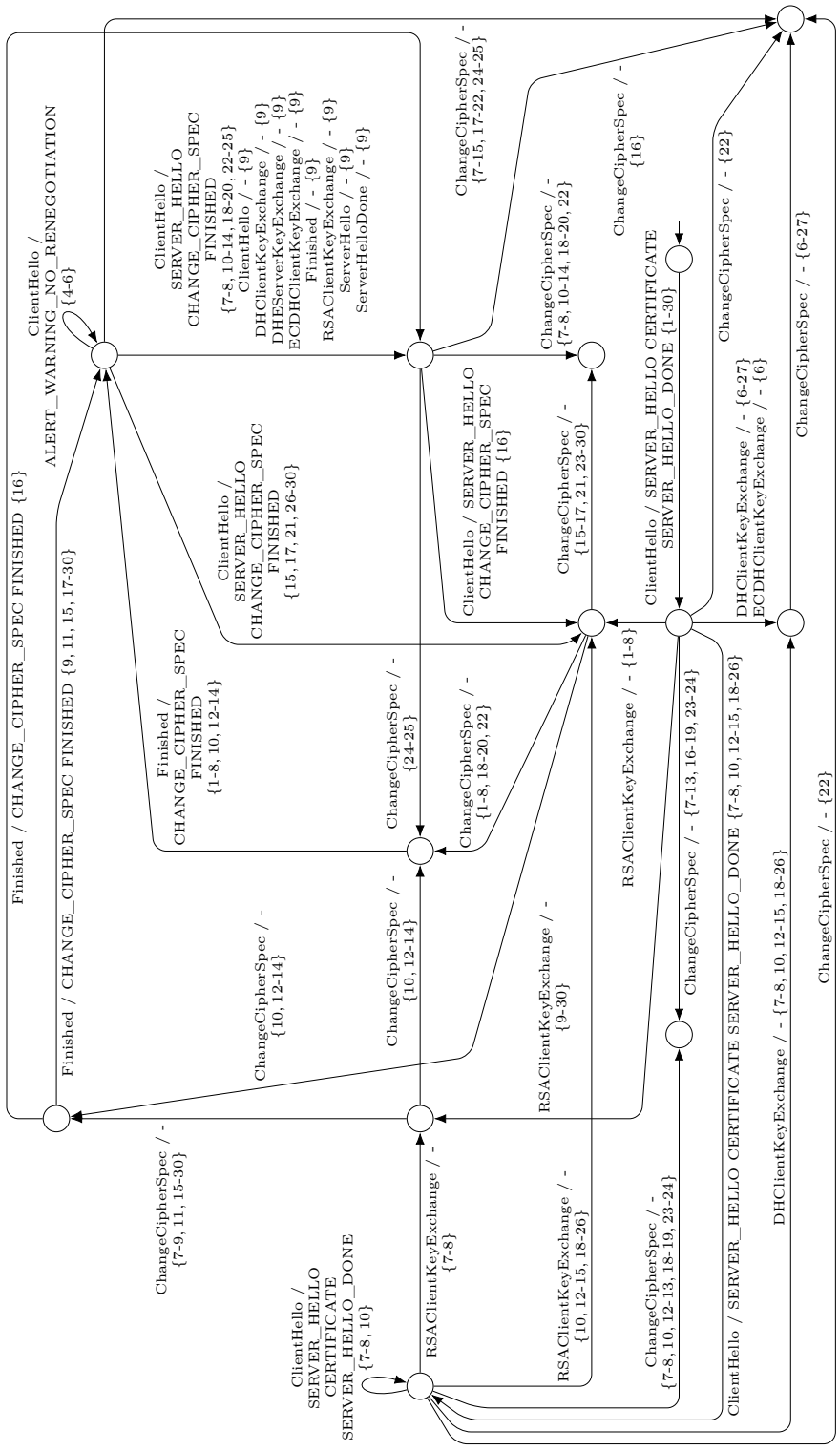


Figure 5.9: The merged gVMM for 597 TLS models, produced using gLTSdiff.

Set	Models	Initial	Accepting	EFA	Skip	Tangle
1	913	+00.00%	+00.00%	+00.00%	−02.39%	+00.00%
2	1,071	+00.00%	+00.00%	−31.86%	−01.24%	+00.31%
3	92	+00.00%	+00.00%	−23.85%	−00.57%	+00.00%
4	350	+00.00%	+00.00%	−26.77%	−01.86%	+00.00%
5	87	+00.00%	+00.00%	−45.32%	−02.38%	+00.00%
6	82	+00.00%	+00.00%	−13.27%	−02.97%	+00.61%
7	195	+00.00%	+00.00%	−02.27%	−00.10%	+00.03%
Total	**2,790**	**+00.00%**	**+00.00%**	**−18.34%**	**−01.68%**	**+00.14%**

Table 5.2: Results for the 'reduced number of differences' experiments.

5.5.4 Reduced number of differences in comparison results

To study the effect of various gLTSdiff improvements compared to LTSDiff, we use 7 ASML model sets with before and after models (2 to 6,538 states per model), for a total of 2,790 gDiffEFA comparisons. We start with a baseline configuration, mimicking LTSDiff. For each difference model we measure the fraction of the model that shows differences, by calculating the number of added and removed (parts of) state and transition properties as a fraction of the total number of (parts of) state and transition properties. Then we add our improvements, one by one, considering initial states for scoring, allowing merging of states only if they agree on acceptance, comparing EFA transitions by their parts rather than in their entirety, rewriting skip patterns, and rewriting tangle patterns. For each improvement, we calculate the increase of the fraction, with respect to the baseline for the first improvement, and with respect to the previous improvement for the other improvements.

Table 5.2 shows the results. For these model sets, the initial and accepting state improvements have no effect. We conjecture this is due to the used model inference approach (see Chapter 3), which produces models with a single initial and accepting state, with per definition the same incoming and outgoing transitions. For other models, in our earlier work, we did see improvements. Comparing parts of EFA transition properties leads to significant reductions for most model sets. Model set 1 has guards and updates that are either the same for the before and after models, or are part of behavior that is completely missing for one of them. Hence the 0% improvement. Skip pattern rewriting reduces the differences as well. Tangle rewriting leads to slightly more differences, as expected, since each unchanged/black tangle state is split into two states, one added/green and one removed/red. However, for the 9 out of 2,790 comparison results with tangles, it is our opinion after manual inspection that the rewritten results are more 'intuitive', which we want to define formally in future work.

5.5.5 Effort/quality trade-off

To study the trade-off between computational effort and quality of the result, we apply gLTSdiff to the 2,790 pairs of gDiffEFA models from Section 5.5.4, with all improvements discussed in that section. We use the LTSDiff landmark-based matcher ('ltsdiff'), Kuhn-Munkres score-minimization matcher ('k-m'), and dy-

namic matcher ('dynamic', uses 'k-m' for gLTSs with $\leq$ 45 states and 'ltsdiff' otherwise), as well as two local scorers with 1 and 5 refinements ('local-1' and 'local-5'), the optimized global scorer ('global-opt'), and dynamic scorer ('dynamic', with 'global-opt' for $\leq$ 45 states, 'local-5' for $\leq$ 500 states, and 'local-1' otherwise). With 3 matchers and 4 scorers, we get 12 configurations. To ensure a fair comparison between configurations, we omit 5 very large model pairs. For these 5, comparison with 'global-opt' hits our configured timeout of an hour, or state-pair relations can not be represented due to hitting Java array size limits[3]. With 12 configurations, 2,785 model pairs (with 2 to 1,882 states per model), and 10 runs per model pair, we get 334,200 comparisons in total. We measure, for each configuration, 1) the total running time to perform all 2,785 comparisons averaged over the 10 runs, and 2) the average of the difference fractions (see Section 5.5.4) that result from those comparison runs.

Table 5.3 shows the results. The first four columns indicate the used matcher, used scorer, average running time, and average difference fraction. The last two columns show the time and diff fraction factors, with the best configuration per column getting factor 1, and the other configurations (rows) getting the number of times that their values are higher. So, there are 12 time values in the third column (v_1, v_2, ..., v_{12}). If v_{min} is the minimum of these values, then the time factor for value v_i ($1 \leq i \leq 12$) is computed as $\frac{v_i}{v_{min}}$. These time factors are shown in the fifth column. Similarly, for the diff fractions of the fourth column, the diff fraction factors are shown in the sixth column.

The first and third configurations should be avoided, as they have significantly worse diff factors than the other configurations. The 'ltsdiff' matcher has good diff factors regardless of the scorer that is used. Somewhat surprisingly, 'ltsdiff/local-1' scores quite well in both time and diff factors. It seems to be a good choice if results are to be obtained quickly. If a bit more time can be spent, 'ltsdiff/dynamic'

[3]The experiments in this chapter were performed using gLTSdiff v1.0.1. In gLTSdiff v1.1.0, a different matrix representation is used to resolve these issues. Rather than hitting Java array size limits, we then run out of memory.

Matcher	**Scorer**	**Time [s]**	**Diff fraction**	**Time factor**	**Diff factor**
dynamic	local-1	152.38	0.07312	1.00000	1.45550
ltsdiff	local-1	161.84	0.05073	1.06204	1.00985
k-m	local-1	174.94	0.09127	1.14803	1.81691
dynamic	dynamic	213.42	0.05068	1.40056	1.00878
dynamic	local-5	221.67	0.05075	1.45466	1.01022
k-m	dynamic	226.44	0.05303	1.48595	1.05562
ltsdiff	dynamic	228.57	0.05059	1.49994	1.00704
ltsdiff	local-5	233.02	0.05062	1.52915	1.00765
k-m	local-5	234.74	0.05216	1.54046	1.03823
dynamic	global-opt	2,335.78	0.05069	15.32817	1.00898
k-m	global-opt	2,369.89	0.05023	15.55203	1.00000
ltsdiff	global-opt	2,493.60	0.05060	16.36385	1.00724

Table 5.3: Results for the trade-off experiments, sorted on time.

seems a good choice, as it has the second-best diff factor. 'k-m/global-opt' has the best diff factor, but this comes at a significant time cost.

We also ran these experiments with a non-optimized global scorer. Since it only affects running time, the diff fractions are the same as for the optimized global scorer. Instead of 5 model pairs then 68 model pairs hit timeouts and Java array size limits. Excluding these 68 model pairs, the optimizations make the global scorer approximately 99 to 101 times faster.

5.6 Conclusions and future work

In this chapter we introduce the gLTSdiff framework, a generalization and extension of the LTSDiff algorithm. It addresses the challenges from Section 5.2. The configurable state and transition properties, as well as configurable, reusable and composable combiners, and the extensible library, allow comparison of different kinds of models, in different ways. gLTSdiff therefore provides an extensible, configurable, scalable approach to compare a variety of models, for a variety of applications.

However, we have only begun to explore the full possibilities that our framework unlocks. It is future work to compare other kinds of models, such as richer EFAs, Timed Automata and hierarchical StateCharts. Other future work includes considering further rewriters, and extending existing rewriters to other model types. Furthermore, the different matchers and scorers could be evaluated on other models, and the gLTSdiff framework could be applied and evaluated for other applications. Finally, a formal notion of 'intuitiveness' could be defined.

Chapter 6

The MIDS tool

The model inference and model comparison approaches outlined in this thesis together form the MIDS methodology (see Section 1.4). The MIDS tool, the *Model Inference and Differencing Suite* [120], implements the MIDS methodology. It offers automation, and therefore provides an efficient way to apply the methodology, reducing the effort that engineers have to spend. The tool is open source, allowing it to be widely used.

Figure 6.1 shows an overview of the functionality provided by the MIDS tool. It closely resembles the overview of the MIDS methodology, but there is one major difference: the MIDS tool features only a single model inference technique, while the MIDS methodology features two such techniques. The Constructive Model Inference (CMI) approach, our state machine learning approach that infers models from execution logs (see Chapter 3), was successfully applied to infer the behavior

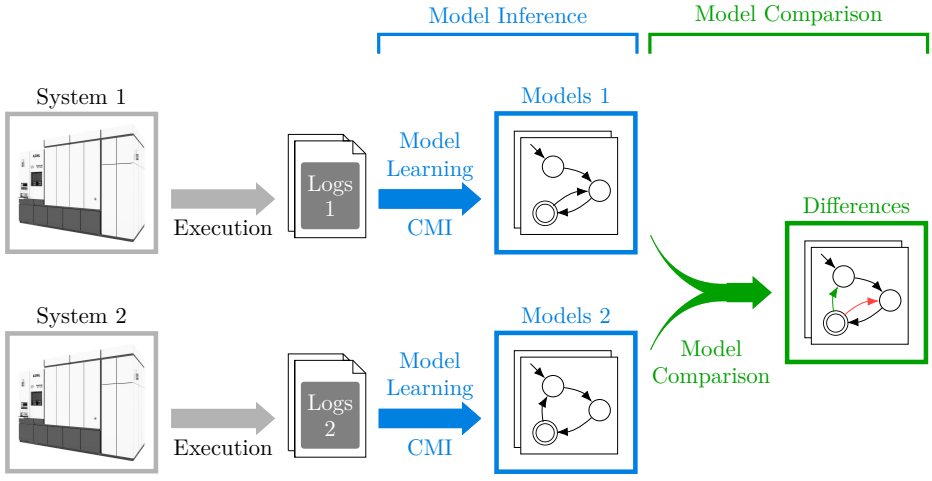


Figure 6.1: An overview of the MIDS tool, consisting of Model Inference (the blue parts, in the middle) and Model Comparison (the green parts, on the right).

of large industrial component-based systems (see Sections 3.5, 4.3 and 5.5). CMI is therefore included in the MIDS tool. Active automata learning as a model inference technique is not included in MIDS. It is still too cumbersome to set up a proper learning setup, and the technique still suffers from scalability issues, making it unsuited for application at the scale we consider (see Section 2.5).

In this chapter, we take a closer look at the MIDS tool, from the perspective of applying the tool in industrial practice, to create impact with the MIDS methodology. We first look at two examples of the tool in action, based on small ASML case studies, to show how the MIDS methodology is worked out in the MIDS tool, as well as how the MIDS tool differs from the MIDS methodology (Section 6.1). And then we look at the extensibility of MIDS, and how it could be used at other companies, as well as for software that is not component-based (Section 6.2).

6.1 MIDS in action: Two examples

We look at two examples of the MIDS tool in action. The first example shows model inference, using our CMI method (see Chapter 3). We especially look at the visualization of the resulting multi-level model. The second example shows model comparison, using our multi-level comparison approach (see Chapter 4) and gLTSdiff structural comparison approach (see Chapter 5). We especially look at the generated comparison report.

6.1.1 Example 1: Model inference and visualization

In the first example, we showcase model inference using the MIDS tool. We in particular discuss the visualization of the resulting multi-level model, which is a unique feature of the tool, not yet described in Chapter 3.

To start, we execute an existing system acceptance test, and capture all inter-component communications in an execution log. From this execution log, we create a Timed Message Sequence Chart (TMSC), as defined by Jonk et al. [74] (see Chapter 3). This functionality is not provided by the MIDS tool itself, but by the *Platform Performance Suite* (*PPS*) [121]. PPS is an open source tool, and it is included in the MIDS tool. The MIDS tool itself then takes the TMSC as input. The TMSC is thus the starting point for the model inference using the CMI method.

We apply CMI and obtain a multi-level model of the communications between the various components. We skip Step 5 of the CMI algorithm (see Section 3.3.5), and thus do not inject stateful behavior. Instead, we generate a multi-level visualization from the inferred multi-level model. The generation of the visualization is part of the MIDS tool. The tool generates a diagram that can be opened with the yEd graph editing software[1].

We focus here on the communications between only two components. These two components have only a few interactions, but that is sufficient to show the features of the visualization. We thus filter the multi-level model to only those service fragments of the two components that are involved in communications between the two components.

[1] See `https://www.yworks.com/products/yed`.

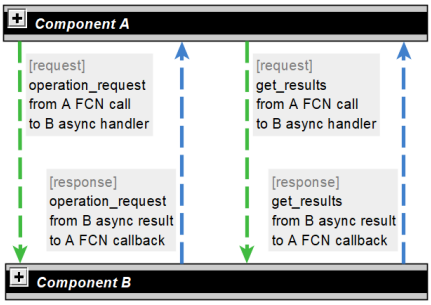


Figure 6.2: System level visualization for Example 1 (anonymized).

System level: Figure 6.2 shows the visualization at the system level. We see the two components, *A* and *B*. Component *A* sends an *operation_request* request to component *B*. This request is sent as an FCN call, an asynchronous request (see Section 2.3.1). The request is handled by component *B* in an asynchronous handler. Component *B* responds to this request by invoking the asynchronous result function for the request, which leads to an FCN callback handler being triggered on component *A*. Component *A* later sends a *get_results* request to component *B*, again using an FCN call, and this is again handled by component *B* using an asynchronous handler. Component *B* responds by invoking the relevant asynchronous result function and this again leads to an FCN callback handler being triggered on component *A*. We thus see two communications from component *A* to component *B*. In principle, at this level, we do not know the order between the different requests. For that, we need to zoom in further, to the next level.

Component level: To go to the next level, we can expand the components, by clicking their '+' icons. This will reveal the internals of the components. Figure 6.3 shows the visualization at the component level. We again see the two components, *A* and *B*, but now in expanded state. We also still see the four possible interactions between the two components, as arrows between the larger boxes. Within each component, we now see service fragments.

In this example, we see that component *A* in its *load* service fragment sends the *operation_request* request to component *B*, that handles this in its *operation_request* handler service fragment. Even though this is an asynchronous handler, the asynchronous result is sent back synchronously in that same service fragment. Component *A* handles the response in its *operation_request* callback service fragment. There it sends the second request, the *get_results* request, to component *B*. Component *B* handles this request in its asynchronous handler service fragment *get_results*. This time, it does not synchronously respond back to component *A*. Instead, it sends the response at a later time in service fragment *results_available*. Component *A* then handles that response in its *get_results* callback service fragment.

The requests from component *A* to component *B* are indicated by green arrows. The corresponding handler service fragments that handle the requests are indicated by green boxes. Similarly, responses from component *B* to component *A* are indicated by blue arrows, and the corresponding handler service fragments that

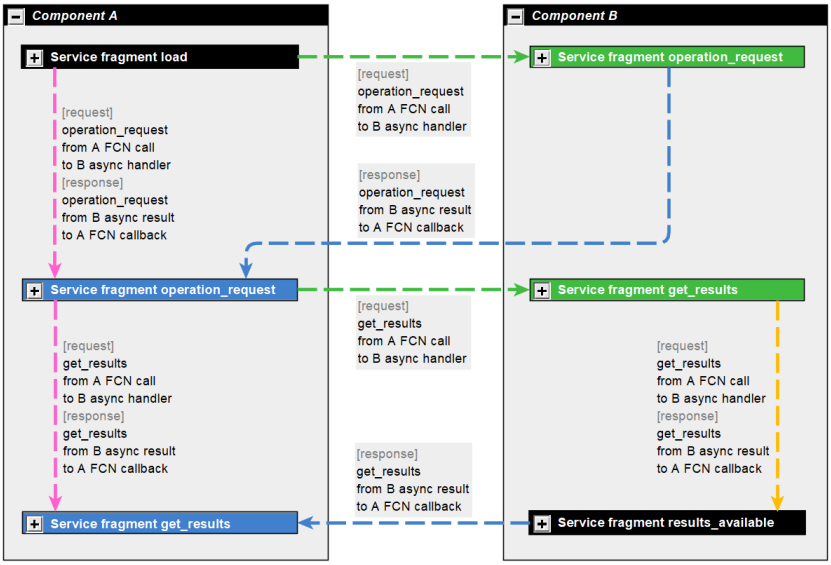


Figure 6.3: Component level visualization for Example 1 (anonymized).

handle the responses are indicated by blue boxes. Two of the service fragments are indicated by black boxes, indicating them to be internal service fragments, that do not handle requests or responses. Given the filtering that we applied to scope to only these two components, all communications with other components are omitted from the diagram. Hence, in this scoped diagram, these two service fragments no longer correspond to requests or responses as they did in the full scope, and are considered internal service fragments for this particular view of the software behavior.

Besides the green and blue arrows, the diagram also features purple and yellow arrows. A purple arrow connects two service fragments on the same component, where the service fragment at the start of the arrow sends a request, and the service fragment at the end of the arrow handles the response to that request. If a single service fragment both sends a request and handles the response, the purple arrow is omitted. Similarly, a yellow arrow connects two service fragments on the same component, where the service fragment at the start of the arrows handles a request, and the service fragment at the end of the arrow sends the response to that request. If a single service fragment both handles a request and also sends the reply, the yellow arrow is omitted.

With these various arrows, we can follow the execution and communication flows through the software. For instance, we can follow the communication flow, by following the arrows from service fragment *load* in A, to *operation_request* in B, to *operation_request* in A, to *get_results* in B, to *results_available* in B, and to *get_results* in A. Similarly, we can follow the execution flow between service fragment in A, by following the arrows from *load* to *operation_request* and *get_results*. We can thus observe the order of the communications and service fragment executions.

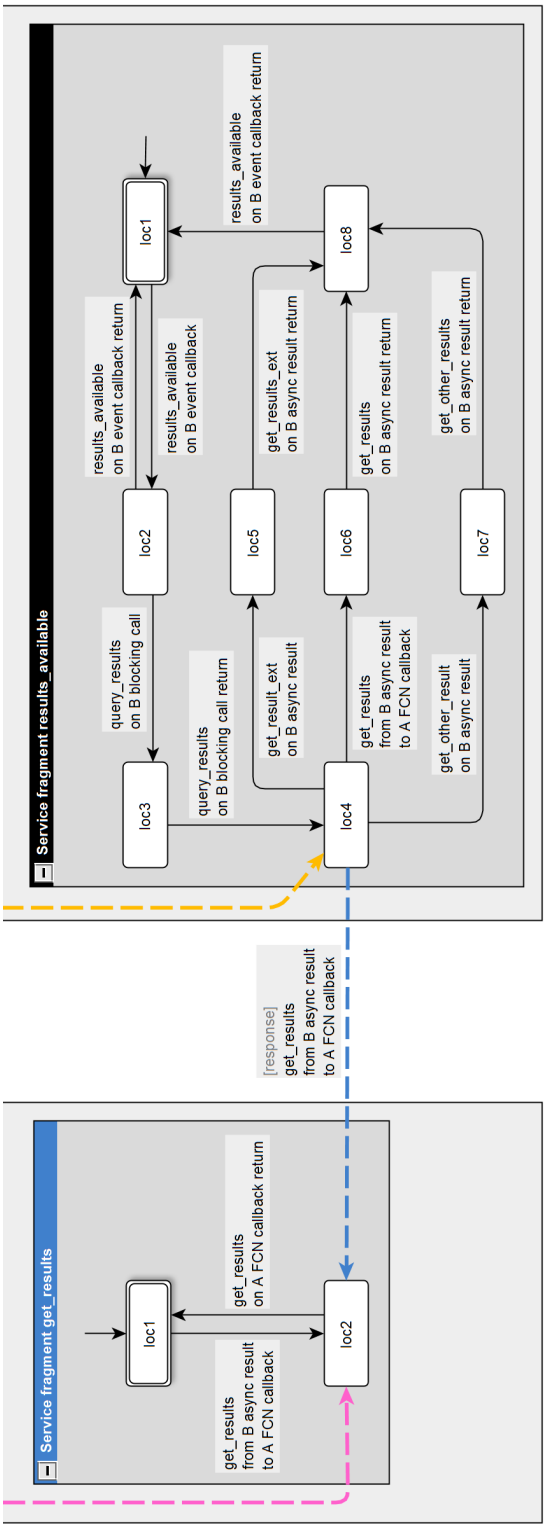


Figure 6.4: Partial service fragment level visualization for Example 1 (anonymized).

Service fragment level: There are however aspects of the behavior that we can not see at the component level. For instance, we can not see under which circumstances the response from component B to component A is sent in service fragment *results_available*. For that, we need to zoom in even further. Similarly to how we can expand components to reveal their internals, we can also expand service fragments.

We expand service fragments *results_available* on component B and *get_results* on component A. Figure 6.4 shows the part of the visualization at the service-fragment level for these two service fragments. Within each service fragment, the state machine of the observed behaviors of that service fragment is shown. For *results_available* we see from the loop between *loc1* and *loc2*, that the callback can start and end, without any other communications taking place in between. Alternatively, it is possible that other communications do happen in between. Component B may perform a blocking call *query_results*, going to *loc3* and *loc4*. There, three different things may happen. Either component B sends a response for the *get_results* request that we discussed earlier, and goes to *loc6* and *loc8*. Alternatively, it may send responses for two other requests, namely for *get_result_ext* (*loc5* and *loc8*) and *get_other_result* (*loc7* and *loc8*). We did not see the start of the asynchronous handling for these other two requests, as they were filtered out. Regardless of which asynchronous response is sent, at the end the state machine finishes when the *results_available* callback returns, and component B becomes idle again. The *get_results* callback service fragment in component A similarly has a state machine. This state machine is much simpler. We only see that the callback starts and returns. No other communications have been observed from this service fragment. Likely, it only updates some internal state.

We can thus see the detailed behavior of the service fragments as depicted by their state machines. The service-fragment-level view also provides additional context to the communications we already observed at the higher level. We can see where component B sends the *get_results* response. By following the yellow arrow, we can see it goes to *loc4*, where the response is being sent. In *loc4* we see two outgoing arrows labeled with *get_results*. We see a black arrow, showing how the state machine transitions to *loc6* when sending the response. And we see a blue arrow, showing the communication to component A. Similarly, in component A, we see multiple related arrows. We see the blue arrow there as well. We also see a purple arrow. It indicates where in the state machine the response is received. We also see a black arrow, showing how the state machine transitions from *loc1* to *loc2* when the response is received and the *get_results* callback is called.

Conclusion: By means of the system-level, component-level, and service-fragment-level views, the multi-level visualizations allow to interactively zoom in, and step-by-step inspect more detailed information. Engineers can follow the interactions of the components, and the execution of the software. This gives them insight into the software behavior, to understand how the software currently works, which is essential to make correct software changes.

6.1.2 Example 2: Model comparison

In the second example, we showcase model comparison using the MIDS tool. We in particular discuss the comparison report that is generated as the result of a

comparison. Rather than discussing a new case study, we look again at the legacy component technology migration case study from Section 4.3.1.

Similar to the first example, we create a TMSC for each execution log. In this case, we have six execution logs, as we have a legacy implementation and a new implementation, and for each implementation we execute three test sets. Then we apply CMI to each of the six execution logs to obtain six model sets. On the six model sets, we apply our comparison approach.

The MIDS tool allows comparison at different levels of the multi-level models inferred by CMI, namely at the service-fragment level (such as in Section 4.3.1), and the component level (such as in Section 4.3.2). Component models are formed from service-fragment models, and thus they contain the same information, at different levels of granularity (see Section 3.3.4). We typically compare models at the most fine-grained level, the service fragment level. This provides us with the most detailed comparison. It also means that we compare more models, as each component typically consists of multiple service fragments. However, the service-fragment models themselves are smaller (have less states). Smaller models lead to faster comparison. With smaller models the comparison results are also smaller, and therefore typically easier to inspect and analyze, especially the structural comparison results. If the scope of the comparison is much larger, we may not be interested in all of the details, and we may instead opt to compare at the component level. For instance, we compared at the component level when we compared the system behavior for various recipes (see Section 4.3.2). For the legacy component technology migration case study, we have a rather limited scope, namely the legacy component and its neighbors. In this case, we thus compare at the service-fragment level.

The MIDS tool produces a comparison report as the output of a comparison. This report is an HTML report, that can be opened in a browser. The report can thus be inspected without requiring the MIDS tool to be installed. The HTML report can also easily be shared.

Figure 6.5 shows the main overview page of the comparison report generated by the MIDS tool, for the legacy component technology migration case study. The overview clearly shows the six levels of the comparison results. The six levels are divided into two groups: for levels $1 - 3$ model sets are compared, and for level $4 - 6$ models within the model sets are compared. Each level is shown, with its name and a short description. For each level, the blue *View* button can be used to inspect the results for that level.

We start by inspecting the results for level 1, shown in Figure 6.6. The results match the output of Figure 4.6 in Section 4.3.1. We then inspect the results of level 4, shown in Figure 6.7. This overview is slightly different than the one in Figure 4.6b. In the output of the MIDS tool, we add suffixes to variant identifiers, per entity/row, to make it clearer to users that variants are assigned per entity.

Finally, we inspect the results of level 6, for the *apply* service fragment, shown in Figure 6.8, and for the *prepare* service fragment, shown in Figure 6.9. The results differ slightly, as here we used gLTSdiff rather than LTSDiff, and we enabled post-processing (see Section 5.3.7). Also, we now use arrows labeled with '...' to indicate abbreviations.

The comparison report has a few other features. It has an *About* page, see

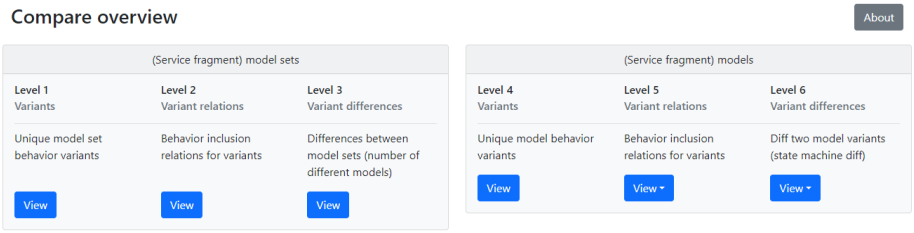


Figure 6.5: Comparison report overview page for Example 2.

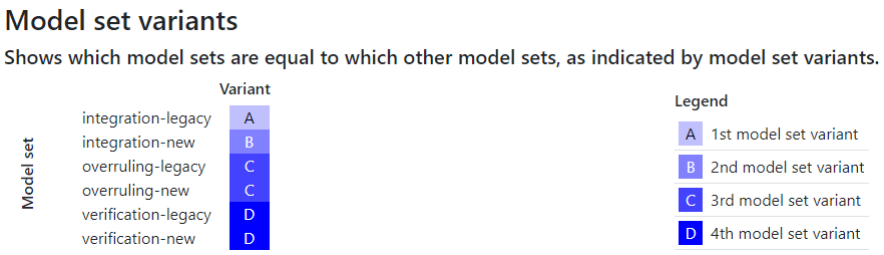


Figure 6.6: Comparison report level 1 page for Example 2 (anonymized).

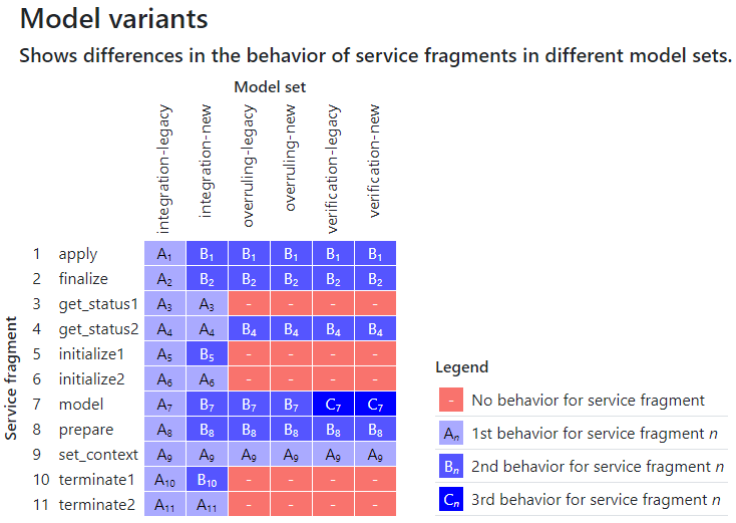


Figure 6.7: Comparison report level 4 page for Example 2 (anonymized).

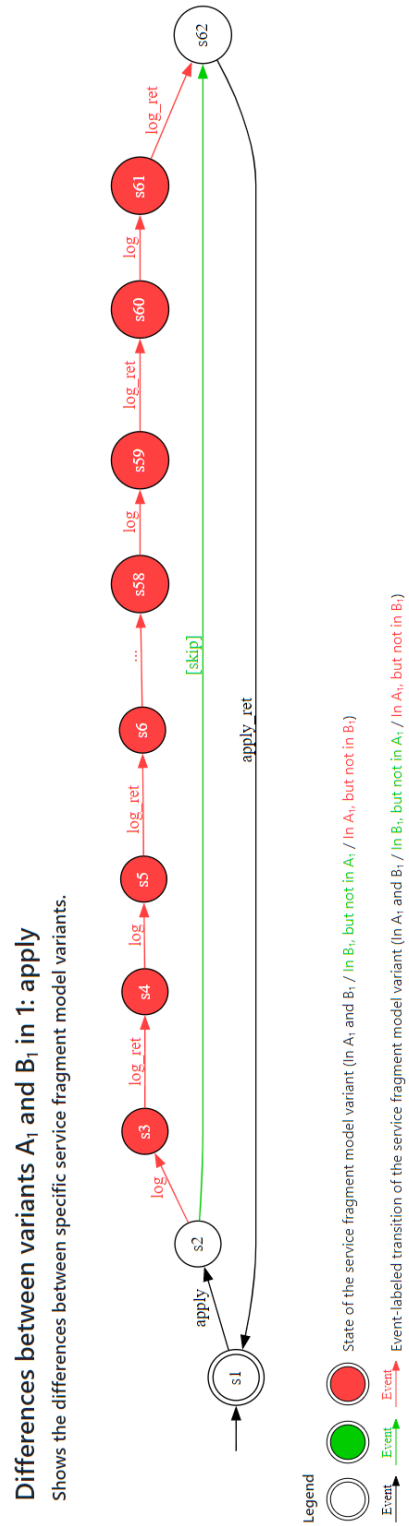


Figure 6.8: Comparison report level 6 page for the *apply* service fragment of Example 2 (condensed and anonymized).

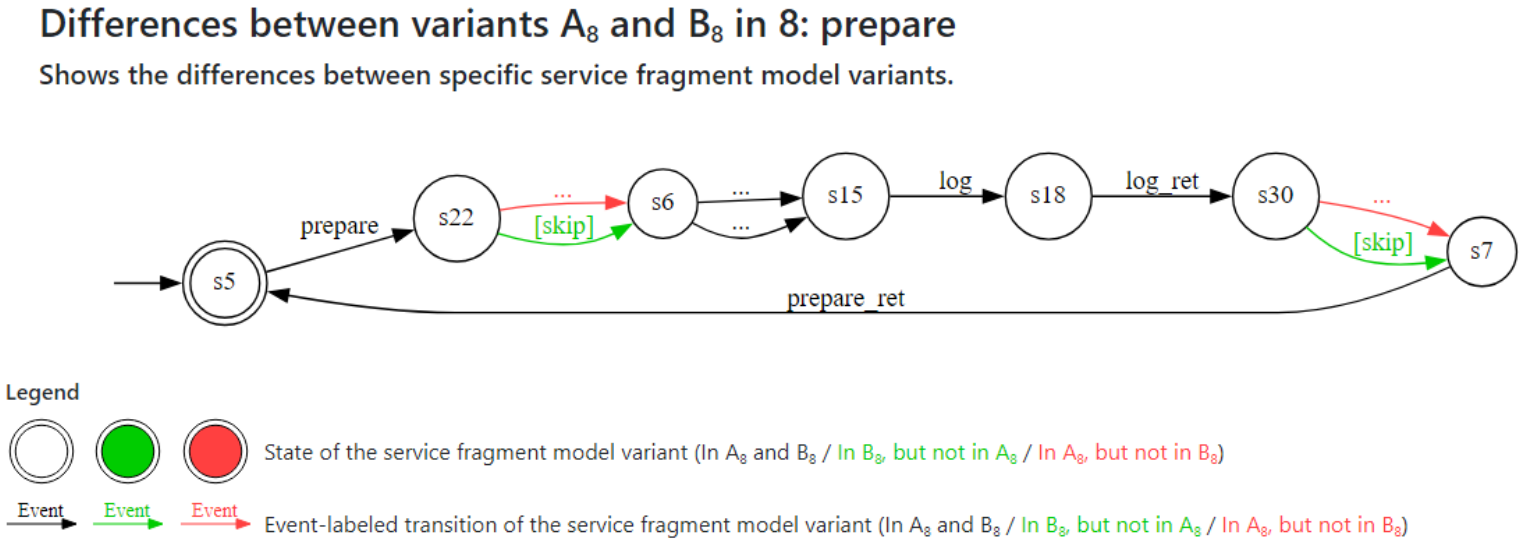


Figure 6.9: Comparison report level 6 page for the *prepare* service fragment of Example 2 (condensed and anonymized).

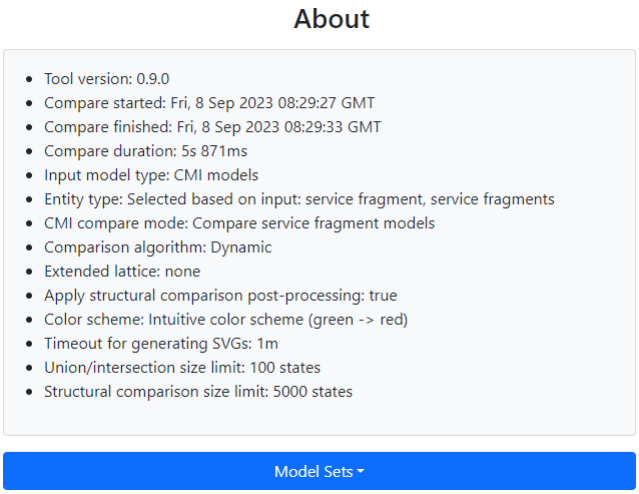


Figure 6.10: Comparison report *About* page for Example 2.

Figure 6.10. It shows information about the configuration used for the comparison, and the model sets that were used as input. The comparison report also allows to inspect the input models, in a similar way as the structural comparison results are shown in level 6.

The information contained in the comparison report allows engineers to inspect behavioral differences at various levels of abstraction, zoom in on the parts of the system that have differences, and determine whether the differences are expected or not. This way, they either find regressions, or they increase their confidence in the correctness of their changes. This reduces the risks for software evolution.

6.2 Extending MIDS for use at other companies

The MIDS methodology is generic. It is not specific to ASML, and can thus also be applied at other companies. To enable this, we open sourced the MIDS tool [120]. It is available under the MIT license, allowing everyone to use it, even commercially.

6.2.1 Extensions of MIDS

While the MIDS tool is thus open source and generic, this does not mean that any engineer can just download the tool and use it as is. The tool may need to be extended to fit particular companies. The MIDS tool provides clear interfaces that allow the tool to be extended.

The execution logs that are used to construct the TMSC models, which are the input to the CMI approach, are typically different for each company. What should be captured in the execution logs, and how this information should be used to construct TMSC models, is not specific to MIDS. The TMSC metamodel is defined by the PPS tool [121]. Concrete TMSC models are the input to the MIDS

tool, and the TMSC metamodel thus serves as an interface for the MIDS tool. Information on how to work with the PPS metamodel and create TMSC models, is described in a PPS report [122].

The TMSC metamodel itself also allows for extensions. Companies can encode their own architectures and communication patterns. Since TMSC models will then be partly specific for each company, the MIDS tool requires a small adapter for each company-specific TMSC dialect. These adapters, called CMI *preparers*, prepare the TMSC by adding standardized properties to various objects in the models. This way, the company-specific architecture information is translated into standardized information for the MIDS tool to work with. The MIDS tool features a built-in CMI preparer that is suited for simple use cases only. The ASML version of the MIDS tool, which extends the tool for ASML, comes with multiple dedicated CMI preparers for some of ASML's company-specific execution log formats.

While the TMSC metamodel and CMI preparers form the main interface to allow company-specific use of the MIDS tool, there is one other important interface that allows for extensions of the tool. The CMI approach allows for various post-processing operations. This includes step 5 of the CMI approach, to inject stateful behavior and domain-specific knowledge into the models (see Section 3.3.5). It also supports various other operations, such as filtering, renaming, and repetition detection [98]. Post-processing is set up in a modular way. Besides the built-in operations, it is possible to add additional ones. For instance, the ASML version of the MIDS tool comes with two additional post-processing operations, that configure existing operations with ASML-specific configurations.

The compare part of the MIDS tool currently does not feature any extensibility. It can be used out of the box.

6.2.2 Application of MIDS to ComMA

To show that the MIDS tool can also be applied outside ASML, we apply the tool in the context of ComMA [79, 80]. ComMA stands for Component Modeling and Analysis. ComMA is a domain-specific language for the specification of software interfaces, by means of signatures, protocols, and timing and data constraints. The ComMA language and tools are part of the Eclipse CommaSuite™ open-source project[2].

ComMA has its own trace format [21]. We made a translation of this format to TMSCs. With this prototype translation, and the simple built-in CMI preparer of MIDS, we can use ComMA traces as input for CMI.

Eclipse CommaSuite ships with several examples, including a 'Vending Machine' example, which is described in more detail in the CommaSuite tutorial [45]. We use a single ComMA trace for our experiment, and translate it to a TMSC. We use that TMSC as input for CMI. Figure 6.11 shows a part of the resulting multi-level model, with the vending machine itself, the coin checker, and the user.

The prototype translation that we used to translate the ComMA trace to a TMSC is far from perfect. The input ComMA trace that we used is also far from

[2]See `https://eclipse.dev/comma`. 'Eclipse', 'Eclipse CommaSuite' and 'CommaSuite' are trademarks of Eclipse Foundation, Inc.

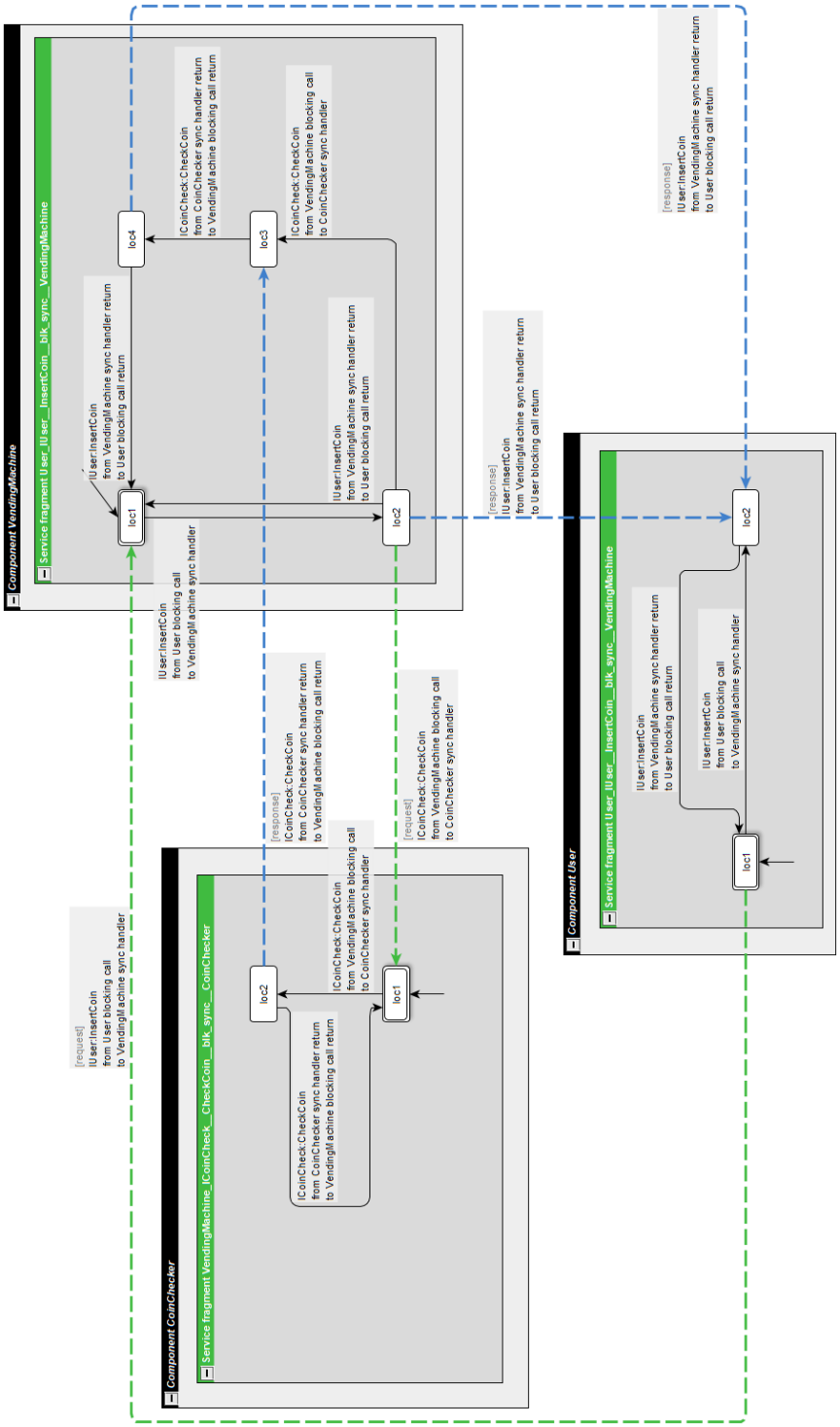


Figure 6.11: Multi-level model for the ComMA 'Vending Machine' example, inferred by the CMI approach of MIDS.

complete. Still, this experiment shows that the MIDS tool can be made to work for other types of execution logs.

6.2.3 Application of MIDS to C code, with timing analysis

Another example of how the MIDS tool can be used with other types of tracing, is the work of Aaron Hilbig [61]. Hilbig extended the MIDS methodology with comparison of timing behavior. To validate his work, he observed the execution of the 'cURL' open source tool, after instrumenting it using certain flags of the GCC compiler.

cURL is not component-based software, but rather a collection of C code files. To apply MIDS, different compilation units are considered as components, and calls between the different compilation units are interpreted as communications between components.

The extension of MIDS with timing analysis was applied to analyze the timing behavior of using cURL to download a file from the Internet. Two different scenarios were executed, where the same file is downloaded using either a cellular connection or a WiFi connection. The expected increase in network response time for the cellular connection over the WiFi connection was apparent in several 'service fragments' (functions), particular those in 'components' (compilation units) related to executing network requests and polling.

6.2.4 Concluding remarks

Currently, the MIDS tool is being rolled out at ASML, with the primary focus on preventing regressions after software changes. The tool is being integrated into the company's systems and way of working, making MIDS widely available for ASML (software) engineers. This will allow MIDS to have a larger impact.

Now that the MIDS tool is open source, we look forward to apply it also at other interested companies. Then, we can also do a proper evaluation of how easy it is to apply MIDS methodology in different settings, and how suitable the resulting models are to address their software evolution challenges.

Chapter 7

Conclusions and future work

To conclude this thesis, in this final chapter we first answer the research questions. And then we reflect further on the work, by discussing some of its implications and shortcomings, which lead to future work. For each research question, we first summarize our main contributions, before answering the question.

7.1 Answering the research questions

We first answer the research questions of this thesis (see the overview in Figure 1.5). We start from the most detailed questions, working our way towards higher-level research questions, and finally the main research question.

RQ-1a How can we efficiently set up and apply active automata learning, to automatically infer the software communication behavior, for large component-based systems?

To answer this research question, we investigated approaches to isolate component code and connect it to an active automata learning tool. Since no systematic approaches existed, we developed a systematic approach to connect software components operating under the client/server paradigm to a learning tool (see Chapter 2). Our general, reusable and configurable framework allows to more quickly produce an active learning setup for such components, dealing with the various types of (a)synchronous communications. We showed the feasibility and effectiveness of our approach that generates large parts of the active learning setup, by applying it to multiple industrial software components.

While our approach significantly reduces the effort to apply active automata learning for large component-based systems, still too much effort is required, as instantiating the framework takes a few hours per (sub-)component. Furthermore, scalability of the learning technique itself continues to be a major challenge. While active automata learning remains a promising technique, in its current state we find it unsuited for application at the scale of the large component-based systems that we consider.

RQ-1b How can we apply state machine learning, such that it automatically infers good approximations of the software communication behavior, for large component-based systems?

To answer this research question, we investigated existing state machine learning algorithms from the literature, to see how they generalize from observations of the system behavior. We found that the existing algorithms generally use hard-to-configure heuristics, that do not relate to the characteristics of component-based systems. And they often over-generalize beyond the system behavior. We therefore developed our Constructive Model Inference (CMI) approach, which infers multi-level models that match the structure of component-based systems (see Chapter 3). We analyzed several properties of the approach and the way it generalizes the observed behavior, without over-approximating beyond the actual system behavior. The approach is mostly automated, and using an industrial case study we showed that it can be applied to large component-based systems, producing models that engineers were able to interpret.

The CMI approach thus tailors state machine learning to learning the software communication behavior for large component-based systems. It also, for this setting, addresses some of the major limitations of existing state machine learning approaches, namely over-generalization and hard-to-configure heuristics. The CMI approach thereby improves the applicability of state machine in industry, for large component-based systems.

RQ-1 How can we efficiently obtain a complete overview of the software communication behavior, for large component-based systems?

We answer this research question by capturing the software communication behavior in multi-level state machine models. We reduce the laborious and error-prone manual modeling work by to a large degree automating the creation of such models, using our CMI approach.

We used CMI successfully for various case studies throughout this thesis (see Sections 3.5, 4.3 and 5.5). In total, CMI has been applied at ASML in over a dozen real-world cases, either by us or by ASML engineers themselves. The CMI approach is largely automated, and for our use cases it does not require complicated configuration. It also scales well when applied to large systems, as we are able to infer the behavior of all software components of an ASML TWINSCAN system[1], inferring the behavior of hundreds of components, from dozens or even hundreds of millions of events. So far, software engineers, software architects, and component owners were able to interpret the models that result from the CMI approach.

The CMI approach uses execution logs as input, which may not contain all communication behavior of the system. Hence, we can not guarantee a complete overview of the software communication behavior. We further reflect on this in the next section.

RQ-2b How can we present engineers with suitable representations of relevant

[1]We currently only look at the main host of the machine, which contains the majority of the (control) software. The other hosts mostly run embedded software to control lower-level sub-systems. Dealing with other hosts is future work.

behavioral differences, for them to efficiently and effectively find any behavioral regressions?

To answer this research question, we investigated various existing comparison techniques for state machine models, to compare the externally observable behavior of state machines, as well as their internal structure. The techniques can be combined to use their complementary strengths, but a single integrated approach and good overview of the results was lacking. We therefore developed a multi-level approach for behavioral comparison of large component-based systems, which integrates multiple existing complementary methods to automatically compare the behavior of state machine models (see Chapter 4). The comparison results can be inspected at six levels of abstraction, ranging from very high-level differences to very detailed ones. Users are guided through the differences in a step-by-step fashion. At each level the differences are presented with a dedicated visualization, that is tailored to allow engineers to zoom in on the parts of the system with differences, wasting no time on the parts without any differences. We evaluated the approach using multiple industrial case studies, thereby demonstrating that it can be applied to large (sub-)systems, provides engineers insight into the behavioral differences, and allows them to find unintended regressions.

RQ-2b How can we present engineers with suitable representations of relevant behavioral differences, for them to efficiently and effectively find any behavioral regressions?

To answer this research question, we investigated reducing the number of irrelevant differences, such that engineers can more efficiently and effectively inspect the relevant differences. We focused on the most detailed level of our multi-level behavioral comparison approach (level 6), where we found that engineers spend the most effort in interpreting the differences. Level 6 shows the behavioral differences based on the results of a structural comparison of state machines. We improved LTSDiff, an existing state-of-the-art algorithm to structurally compare state machines, by generalizing and extending it to gLTSdiff (see Chapter 5). We applied gLTSdiff to several large-scale industrial and open source case studies. We showed that it efficiently computes behavioral differences for large numbers of input models; that it reduces the number of differences in comparison results, which reduces the effort engineers have to spend to interpret the differences; and that it can be used to effectively find behavioral regressions.

RQ-2 How can we efficiently obtain a complete overview of the system-wide impact on the communication behavior caused by software changes, for large component-based systems?

We answer this research question by combining our multi-level behavioral comparison approach with our structural comparison framework gLTSdiff. The former provides the overview, using the latter to provide the most detailed information. We showed that both are able to efficiently handle large component-based systems, and reduce the effort that engineers have to spend on inspecting and interpreting differences. We further showed that our combined approach can be used to find behavioral regressions. Engineers can thus determine the (undesirable) impact of their software changes on the communication behavior.

RQ How can we reduce the efforts and risks of software evolution, for large component-based systems?

We answer this research question with our MIDS methodology, which combines our model inference and model comparison approaches. Both approaches reduce the risks of software evolution. Model inference allows engineers to infer models of the software that give them insight into the current communication behavior. Understanding the behavior of the software before they change it, aids them in making proper changes to the software. And model comparison allows them to compare the (inferred) software behavior models, zoom in on the relevant differences, and determine the impact of software changes. Together model inference and model comparison assist them in increasing confidence that the software changes were made as intended, preventing regressions, and thereby reducing some of the risks before these changes are delivered.

We rely on automation provided by the MIDS tool, the Model Inference and Differencing Suite, to reduce the effort that engineers have to spend in applying the methodology (see Chapter 6). The tool is also extensible, allowing to support applications at different companies, which may for instance have execution logs in different formats. The tool is open source, allowing wide-spread use of the tool and its underlying methodology in industry, to reduce the effort and risks for software evolution in their large component-based systems.

7.2 Further reflection and future work

Next, we reflect further on the work, discussing its implications, and identifying shortcomings, leading to future work. We discuss the following topics:

1. What is the impact of focusing only on functional software behavior, and not (for instance) on performance aspects?
2. What is the impact of focusing only on the communication behavior of components, and not on their internals?
3. What is the impact of focusing only on the order of the communications, and not (for instance) on data relations?
4. How effective is the approach in dealing with concurrency and asynchronous communications?
5. What is the impact of incomplete execution logs on the completeness of the inferred models?
6. What is the overall impact of the work?

7.2.1 Focus on functional software behavior

We scoped our work by focusing primarily on software functionality, considering the non-functional aspects of software changes to be out of scope. We made this decision from the belief that if we are successful in managing the complexity of functional software changes, we can already have a significant impact on reducing the challenges for software evolution in industry (as outlined in Chapter 1). The

various case studies we performed, and that are described in this thesis, show that we can indeed reduce effort and risks for software evolution. For instance, in Section 5.5.1, we found the cause of an already delivered regression in two hours rather than in two days. And in Section 5.5.2, ASML engineers were able to find a regression before delivery that was not found by existing tests, reducing risks and saving at least a person-day in effort.

However, while non-functional aspects are thus out of scope for this thesis, our approach may at times also be able to provide insights into, and prevent issues relating to, non-functional aspects. This is for instance shown in the two aforementioned case studies from Section 5.5. For the wafer exposure regression case study, we were able to pinpoint the functional change that led to a system throughput regression. And for the wafer stage refactoring case study, an ASML engineer was able to find and fix a mode switching issue that would have otherwise led to a system throughput regression.

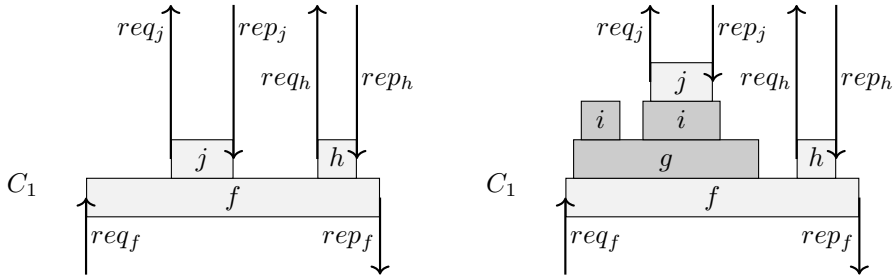
Being able to see some impact on performance by only considering functional changes is nice. But, to thoroughly analyze the software performance, and changes in performance after the software is changed, requires including timing information into our approach. Some first work has already been performed by Aaron Hilbig (see also Section 6.2.3). His work resulted in a prototype implementation of MIDS that takes timing into account in both model inference and model comparison. However, working this out in more detail, maturing it, and including it in the open source MIDS tool, remains future work.

7.2.2 Focus on communication behavior of components

We scoped our work by focusing on the communication behavior between components in a large component-based system. This choice is based on our experience in working with various companies in the Dutch high-tech industry, and the challenges they face (see Chapter 1). We believe that, in such systems, engineers typically already have a good grip on their own component. It is the interaction with the rest of the system, especially the impact on further-away parts of the system, where they lack the insights and the knowledge. The integration of components to form (sub-)systems often reveals issues, both in the individual components and their composition, not in the least due to the complexities of inter-component communication.

However, while that is our focus in this thesis, it does not mean our approach is limited to the externally visible communication behavior of components. We believe our approach could also be applicable to the internal software behavior of the components. If suitable logging is available for the function calls within a component, this could be incorporated into the inferred models. We performed some first experiments, using function-level logging available within ASML.

Figure 7.1 shows two versions of an example TMSC model, where on the left only the externally-observable communications (function calls, light gray bars) are captured, and on the right additionally the internal function calls (darker gray bars) are added. A request to execute function f leads to external calls for functions j and h. If the internal function calls are considered, f first calls g, which calls function i twice. The second call to i leads to the j call. Function h is directly called from function f.



(a) External function calls only

(b) External and internal function calls

Figure 7.1: An example TMSC model, with and without internal function calls.

Each component still consists of well-formed call stacks. Hence, this fits well with our CMI approach, which could be adapted to take the additional calls into account when inferring state machines. And our comparison approach works on state machines, so it can be used without any fundamental changes. First experiments thus show that our approach is not fundamentally limited to considering the externally-visible communication behavior of components, and it could also consider the internal software behavior of components. However, working this out in detail is future work.

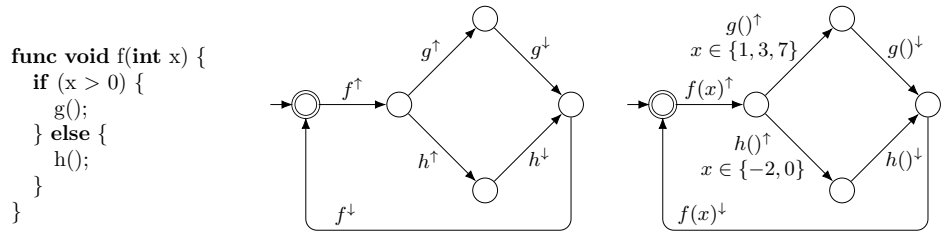
7.2.3 Focus on order of communications

We scoped our work by focusing on the order of communications between components. We thus consider only a part of what comprises a good interface protocol. A good protocol should consist of at least the following aspects [80]:

- The *syntax*, e.g., the functions that may be called, and if relevant also their parameters, and the types of the parameters and return values.
- The *order* constraints, e.g., the order in which the functions may be called. This can for instance be captured as a protocol state machine.
- The *data* constraints, e.g., in which situations a component replies in a particular way, or sends requests to other components, based on the values of the parameters.
- The *time* constraints, e.g., the amount of time a component needs to reply to a request, or the number of requests it can handle in a certain time period.

We primarily consider the first two aspects in this thesis. Our approach can be extended to also consider timing, as discussed in Sections 6.2.3 and 7.2.1.

An important aspect we have not yet discussed is the influence of data. We primarily considered components whose behavior is predominantly determined by the messages they exchange, rather than for instance the data that they process (see Section 1.1.2). However, data can have an impact on the software behavior. Figure 7.2 shows an example. In the code (Figure 7.2a) there is an 'if' statement, where in the 'then' branch one communication takes place, while in the 'else'



(a) Source code (b) Plain state machine (c) Extended state machine

Figure 7.2: An example of the influence of data on the software behavior.

branch a different communication occurs. If we observe both these variants of behavior in the execution logs, our CMI approach will infer a state machine in which both behaviors are possible (Figure 7.2b). From this state machine, it is clear that a choice is possible, but not in which cases which branch is taken. We thus see the effect of decisions that are made based on data, but not the data itself.

We envision our approach could be extended to make the data relations explicit. The arguments of function calls could be recorded in the execution logs. The data could then be taken into account when inferring state machines using our CMI approach. Figure 7.2c shows what the result could look like, if we have several observations for calls to function f, for three positive and two non-positive values for parameter x.

This is however future work, where several research questions would need to be answered, such as:

1. What arguments or combination of arguments are relevant to a choice, and should thus be included on the transitions?

   The event for the start of the function call, such as $f(x)^{\uparrow}$ in 7.2c, has the information about the arguments. Some of the values of the arguments may be used in 'if' statements in the function body. One could check after each event $f(x)^{\uparrow}$ which path in the state machine is taken, and annotate the choices with all the values of all the arguments of the function, such as for $g()^{\uparrow}$ and $h()^{\uparrow}$. However, likely not all arguments are relevant to the choice. The question is then how to determine which arguments, or combinations of arguments, are minimally needed to distinguish the different choices.

2. What is the generalization of the observed values? Can we create a predicate out of it?

   In the code of a C/C++ function, there could for instance be an 'if' statement guard '`x > 0`' to check that x is positive, or '`x % 2 == 0`' to check that x is even. In the approach as suggested above, we add to the choice transitions those observed values of arguments that distinguish the choices (e.g., $x \in \{1, 3, 7\}$). The question is then how we can generalize such collections of values, and for instance infer back the original guard predicates.

7.2.4 Dealing with concurrency and asynchronous communications

One of the major sources of complexity in large component-based systems is concurrency (see Chapter 1). If components execute concurrently, there can be many interleavings, similar sequences of communications with subtly reordered communications. Asynchronous communication can make this even more complex. There are a few ways we deal with this complexity in our approach, both in model inference and model comparison.

In the CMI approach (see Chapter 3), we infer multi-level models: system models, component models, and service fragment models. The CMI approach is based on several assumptions, of which the following two are key in this regard:

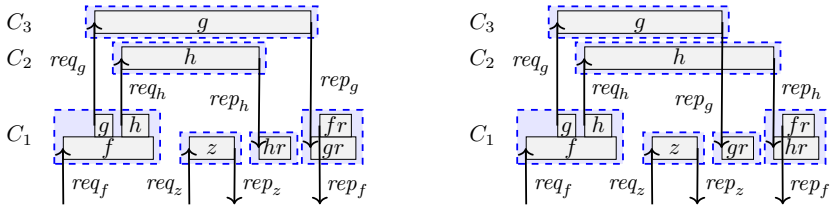
- Components are sequential, e.g., corresponding to a single operating system thread.
- Client requests (and server responses) can only be handled once the component is idle, and prior requests are finished, i.e., service fragments are executed non-preemptively.

This means that concurrency and interleaving are mostly confined to the *system* level (*models*). In system models, the component models are combined, and this is where all the different interleavings of the communications of the concurrently executing components will be fully visible. However, system models are typically used only for analysis, not human inspection. For instance, the complete system behavior may automatically be checked for deadlock or trace conformance, by a computer, as we did in Section 3.5. For human inspection, we do not compute the full system model. Instead, we visualize the multi-level models (see Section 6.1.1). This way, the full system behavior may be inspected, without explicitly showing all the different interleavings.

Stateless *component models* allow their service fragments to be executed in any order. If stateful behavior is injected in step 5 of the CMI approach, stateful component models may see the effects of interleaving. Different service fragments may be executed in different orders, as a result of the environment initiating them earlier or later. Component models are also used primarily for analysis rather than human inspection, and multi-level visualization allows inspecting their behavior without making all interleavings explicit.

Service fragment models are typically not impacted by concurrency and interleaving, as they represent non-concurrent non-interrupted executions of single functions. However, subtle details may still be visible.

For instance, consider Figure 7.3. Figure 7.3a repeats one of the observations for the running example from Chapter 3 (w_1 from Figure 3.3a). In this observation, component C_1 handles req_f by sending two asynchronous requests, req_g and req_h, to two different servers, C_3 and C_2, respectively. Only once both servers have responded can component C_1 reply back to its client by means of rep_f. In Figure 7.3a, rep_h is first and rep_g is second. Therefore, the service fragment that handles rep_g, namely gr, sends rep_f. In Figure 7.3b, instead, rep_g is first and rep_h is second. The order that the replies come in is thus reversed, which could for instance be due to timing differences in the system. And thus the service fragment that handles rep_h, namely hr, sends rep_f. In these observations, rep_f is thus



(a) First observation

(b) Second observation

Figure 7.3: An example of the effects of concurrency on service fragments.

either sent by service fragment *gr* or *hr*, depending on whether rep_g or rep_h comes in last. If only one of the behaviors is observed, then only one of the models for service fragments *gr* and *hr* will include the sending of rep_f. If both the behaviors are observed, then both service fragment models will include it.

There are other, similar subtleties where concurrency and interleaving can pop up in service fragment models. For instance, different clients could send a first request to a component, and the component may behave differently for a first request. Then the service fragments that handle the requests from the different clients may show differences, depending on what is observed.

For *model comparison*, we typically use the service fragment models (see Section 6.1.2), which are least impacted by concurrency and interleaving.

If the subtleties related to concurrency and interleaving are not relevant for a particular use case, the MIDS tool can post-process the models to eliminate such differences. For instance, the identities of different clients could be hidden, or certain service fragments could be merged together. These differences then also do not show up in the comparison results.

Our methodology thus provides several practical ways to cope with concurrency and interleaving. However, it does not completely eliminate their effects.

7.2.5 Completeness of the inferred models

Passive learning approaches, like CMI, infer models from observations. In practice, the observations are typically incomplete. While the approaches generalize beyond the observations, this is often insufficient and thus only a part of the system behavior is captured in the inferred models. Such incompleteness is a fundamental limitation of passive learning approaches.

It is therefore important to observe as much of the system behavior as possible. It can be beneficial to combine observations from different executions. For instance, execute the system in its normal environment, to obtain observations of the regular production behavior. But, also execute various test sets, that test some of the exceptional behavior. Still, it remains difficult to obtain observations of all the relevant system behavior. And at the moment we can give no guarantees on how much of the system behavior we have observed, or inferred.

In the long run, we believe active automata learning can fill the gap. What is not yet observed, could be queried, resulting in more complete observations and thus more complete models.

For now however, we use models that are typically incomplete. Still, in our work, we have applied MIDS to quite some case studies, where we were able to give insight into the system behavior. And we could find regressions that could not be found by existing test suites. Hence, even if the models are incomplete, our approach still has practical value.

7.2.6 The overall impact of the work

MIDS has so far been applied in over a dozen real-world cases, either by us or by ASML engineers themselves. Some of these cases are described in this thesis. We believe this shows the concrete value of the approach for these applications, reducing the effort that engineers have to spend, and reducing the risks for software evolution. It also indicates the potential of the approach in general.

Currently, the MIDS tool is being rolled out at ASML, integrating the tool into the company's systems and way of working. By making the tool widely available, and spreading the word, we hope to increase the number of users, and thereby increase the impact of our work.

Now that the MIDS tool is open source, we look forward to apply it also at other interested companies. Then, we can also do a proper evaluation of how easy it is to apply the MIDS methodology in different settings. Our hope is that we can get MIDS users in various companies, to address their software evolution challenges, and further increase the impact of our work.

Bibliography

[1] W. van der Aalst. *Process Mining: Data Science in Action*. 2nd edition. Springer-Verlag Berlin Heidelberg, 2016. ISBN: 978-3-662-49850-7.

[2] W. van der Aalst, B. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A. Weijters. "Workflow mining: A survey of issues and approaches". In: *Data & Knowledge Engineering*, volume 47, number 2, 2003, pages 237–267. DOI: 10.1016/S0169-023X(03)00066-1.

[3] F. Aarts, F. Heidarian, and F. Vaandrager. "A Theory of History Dependent Abstractions for Learning Interface Automata". In: *Concurrency Theory*. Springer Berlin Heidelberg, 2012, pages 240–255. DOI: 10.1007/978-3-642-32940-1_18.

[4] S. Abbaspour Asadollah, R. Inam, and H. Hansson. "A Survey on Testing for Cyber Physical System". In: *Testing Software and Systems*. Springer International Publishing, 2015, pages 194–207. DOI: 10.1007/978-3-319-25945-1_12.

[5] D. Akdur, V. Garousi, and O. Demirörs. "A survey on modeling and model-driven engineering practices in the embedded software industry". In: *Journal of Systems Architecture*, volume 91, 2018, pages 62–82. DOI: 10.1016/j.sysarc.2018.09.007.

[6] L. Akroun and G. Salaün. "Automated verification of automata communicating via FIFO and bag buffers". In: *Formal Methods in System Design*, volume 52, number 3, 2018, pages 260–276. DOI: 10.1007/s10703-017-0285-8.

[7] B. Aksakal. "Actionable Insights from Software Behavior Model Comparisons for Software Changes in High-Tech Systems". Master's thesis. Eindhoven University of Technology, 2022.

[8] E. S. de Almeida, A. Alvaro, V. C. Garcia, J. C. C. P. Mascena, V. A. de Arruda Burégio, L. M. do Nascimento, D. Lucrédio, and S. R. de Lemos Meira. *C.R.U.I.S.E.: Component Reuse in Software Engineering*. C.E.S.A.R. E-books, 2007.

[9] S. Alrabaee, M. Debbabi, and L. Wang. "A Survey of Binary Code Fingerprinting Approaches: Taxonomy, Methodologies, and Features". In: *ACM Computing Surveys*, volume 55, number 1, 2022, DOI: 10.1145/3486860.

[10] H. Amar, L. Bao, N. Busany, D. Lo, and S. Maoz. "Using Finite-State Models for Log Differencing". In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, 2018, pages 49–59. DOI: 10.1145/3236024.3236069.

[11] D. Angluin. "Learning Regular Sets from Queries and Counterexamples". In: *Information and Computation*, volume 75, number 2, 1987, pages 87–106. DOI: 10.1016/0890-5401(87)90052-6.

[12] K. Aslam. "Deriving Behavioral Specifications of Industrial Software Components". PhD thesis. Eindhoven University of Technology, 2021.

[13] K. Aslam, L. Cleophas, R. Schiffelers, and M. van den Brand. "Interface protocol inference to aid understanding legacy software components". In: *Software and Systems Modeling*, volume 19, number 6, 2020, pages 1519–1540. DOI: 10.1007/s10270-020-00809-2.

[14] ASML. *Customer Support: Supplying the semiconductor industry 24/7*. Accessed: 2023-07-04. URL: https://www.asml.com/en/products/customer-support.

[15] A. Asprovska. *Scalable Model-Driven Software for Stage Alignment of ASML TWINSCAN Machines – Control and Domain Logic Services*. Technical report 2018/095. Eindhoven University of Technology, 2018.

[16] F. Avellaneda and A. Petrenko. "Inferring DFA without negative examples". In: *Proceedings of The 14th International Conference on Grammatical Inference 2018*. Volume 93. PMLR. 2019, pages 17–29. URL: https://proceedings.mlr.press/v93/avellaneda19a.html.

[17] P. Baker, S. Loh, and F. Weil. "Model-Driven Engineering in a Large Industrial Context – Motorola Case Study". In: *Model Driven Engineering Languages and Systems*. Springer Berlin Heidelberg, 2005, pages 476–491. DOI: 10.1007/11557432_36.

[18] R. Baldoni, E. Coppa, D. D'elia, C. Demetrescu, and I. Finocchi. "A Survey of Symbolic Execution Techniques". In: *ACM Computing Surveys*, volume 51, number 3, 2018, DOI: 10.1145/3182657.

[19] L. Bao, N. Busany, D. Lo, and S. Maoz. "Statistical Log Differencing". In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2019, pages 851–862. DOI: 10.1109/ASE.2019.00084.

[20] T. Barik, R. DeLine, S. Drucker, and D. Fisher. "The Bones of the System: A Case Study of Logging and Telemetry at Microsoft". In: *Proceedings of the 38th International Conference on Software Engineering Companion*. Association for Computing Machinery, 2016, pages 92–101. DOI: 10.1145/2889160.2889231.

[21] D. Bera, M. Schuts, J. Hooman, and I. Kurtev. "Reverse Engineering Models of Software Interfaces". In: *Computer Science and Information Systems*, volume 18, number 3, 2021, pages 657–686. DOI: 10.2298/CSIS200131013B.

[22] I. Beschastnikh, Y. Brun, J. Abrahamson, M. Ernst, and A. Krishnamurthy. "Unifying FSM-Inference Algorithms through Declarative Specification". In: *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pages 252–261. DOI: 10.1109/ICSE.2013.6606571.

[23] K. Bogdanov and N. Walkinshaw. "Computing the structural difference between state-based models". In: *Working Conference on Reverse Engineering*. IEEE. 2009, pages 177–186. DOI: 10.1109/WCRE.2009.17.

[24] D. Brand and P. Zafiropulo. "On Communicating Finite-State Machines". In: *Journal of the Association for Computing Machinery*, volume 30, number 2, 1983, pages 323–342. DOI: 10.1145/322374.322380.

[25] H. P. Breivold, M. A. Chauhan, and M. A. Babar. "A Systematic Review of Studies of Open Source Software Evolution". In: *2010 Asia Pacific Software Engineering Conference*. IEEE. 2010, pages 356–365. DOI: 10.1109/APSEC.2010.48.

[26] H. P. Breivold, I. Crnkovic, and M. Larsson. "A systematic review of software architecture evolution research". In: *Information and Software Technology*, volume 54, number 1, 2012, pages 16–40. DOI: 10.1016/j.infsof.2011.06.002.

[27] G. Broadfoot. "ASD Case Notes: Costs and Benefits of Applying Formal Methods to Industrial Control Software". In: *Formal Methods*. Springer Berlin Heidelberg, 2005, pages 548–551. DOI: 10.1007/11526841_39.

[28] G. Broadfoot and P. Broadfoot. "Academia and industry meet: Some experiences of formal methods in practice". In: *Tenth Asia-Pacific Software Engineering Conference*. IEEE. 2003, pages 49–58. DOI: 10.1109/APSEC.2003.1254357.

[29] C. Brun and A. Pierantonio. "Model Differences in the Eclipse Modelling Framework". In: *UPGRADE: The European Journal for the Informatics Professional*, volume 9, number 2, 2008, pages 29–34.

[30] G. Canfora, M. di Penta, and L. Cerulo. "Achievements and Challenges in Software Reverse Engineering". In: *Communications of the ACM*, volume 54, number 4, 2011, pages 142–151. DOI: 10.1145/1924421.1924451.

[31] C. Carrez, A. Fantechi, and E. Najm. "Behavioural Contracts for a Sound Assembly of Components". In: *Formal Techniques for Networked and Distributed Systems (FORTE)*. Springer Berlin Heidelberg, 2003, pages 111–126. DOI: 10.1007/978-3-540-39979-7_8.

[32] C. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. 3rd edition. Springer Cham, 2021. ISBN: 978-3-030-72274-6.

[33] P.-A. Champin and C. Solnon. "Measuring the Similarity of Labeled Graphs". In: *Case-Based Reasoning Research and Development (ICCBR)*. Springer Berlin Heidelberg, 2003, pages 80–95. DOI: 10.1007/3-540-45006-8_9.

[34] E. Chikofsky and J. Cross. "Reverse Engineering and Design Recovery: A Taxonomy". In: *IEEE Software*, volume 7, number 1, 1990, pages 13–17. DOI: 10.1109/52.43044.

[35] C. Y. Cho, D. Babi, E. C. R. Shin, and D. Song. “Inference and Analysis of Formal Models of Botnet Command and Control Protocols”. In: *Proceedings of the 17th ACM conference on Computer and communications security*. 2010, pages 426–439. DOI: 10.1145/1866307.1866355.

[36] R. Cleaveland and O. Sokolsky. “Equivalence and Preorder Checking for Finite-State Systems”. In: *Handbook of Process Algebra*, 2001, pages 391–424. DOI: 10.1016/B978-044482830-9/50024-2.

[37] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. “A Systematic Survey of Program Comprehension through Dynamic Analysis”. In: *IEEE Transactions on Software Engineering*, volume 35, number 5, 2009, pages 684–702. DOI: 10.1109/TSE.2009.28.

[38] I. Crnkovi, S. Sentilles, A. Vulgarakis, and M. Chaudron. “A Classification Framework for Software Component Models”. In: *IEEE Transactions on Software Engineering*, volume 37, number 5, 2011, pages 593–615. DOI: 10.1109/TSE.2010.83.

[39] C. D. N. Damasceno, M. R. Mousavi, and A. da Silva Simao. “Learning by sampling: learning behavioral family models from software product lines”. In: *Empirical Software Engineering*, volume 26, number 1, 2021, pages 1–46. DOI: 10.1007/s10664-020-09912-w.

[40] C. D. N. Damasceno, M. R. Mousavi, and A. Simao. “Learning from Difference: An Automated Approach for Learning Family Models from Software Product Lines”. In: *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A*. Association for Computing Machinery, 2019, pages 52–63. DOI: 10.1145/3336294.3336307.

[41] C. D. N. Damasceno and D. Strüber. “Family-Based Fingerprint Analysis: A Position Paper”. In: *A Journey from Process Algebra via Timed Automata to Model Learning: Essays Dedicated to Frits Vaandrager on the Occasion of His 60th Birthday*. Cham: Springer Nature Switzerland, 2022, pages 137–150. DOI: 10.1007/978-3-031-15629-8_8.

[42] A. van Deursen, E. Visser, and J. Warmer. “Model-Driven Software Evolution: A research agenda”. In: *Proceedings 1st International Workshop on Model-Driven Software Evolution*. 2007, pages 41–49.

[43] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. “Feature location in source code: a taxonomy and survey”. In: *Journal of software: Evolution and Process*, volume 25, number 1, 2013, pages 53–95. DOI: 10.1002/smr.567.

[44] O. al Duhaiby, A. Mooij, H. van Wezep, and J. F. Groote. “Pitfalls in Applying Model Learning to Industrial Legacy Software”. In: *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*. Springer International Publishing, 2018, pages 121–138. DOI: 10.1007/978-3-030-03427-6_13.

[45] Eclipse Foundation. *Eclipse CommaSuite Tutorial*. Accessed: 2023-09-08. 2023. URL: https://eclipse.dev/comma/tutorial/intro.html.

[46] P. Emanuelsson and U. Nilsson. “A Comparative Study of Industrial Static Analysis Tools”. In: *Electronic Notes in Theoretical Computer Science*, volume 217, 2008, pages 5–21. DOI: 10.1016/j.entcs.2008.06.039.

[47] S. Fujiwara, G. van Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. "Test Selection Based on Finite State Models". In: *IEEE Transactions on Software Engineering*, volume 17, number 6, 1991, pages 591–603. DOI: 10.1109/32.87284.

[48] B. Genest, D. Kuske, and A. Muscholl. "On Communicating Automata with Bounded Channels". In: *Fundamenta Informaticae*, volume 80, number 1-3, 2007, pages 147–167.

[49] R. van Glabbeek. "The Linear Time — Branching Time Spectrum II". In: *International Conference on Concurrency Theory*. Springer. 1993, pages 66–81. DOI: 10.1007/3-540-57208-2_6.

[50] R. van Glabbeek and B. Ploeger. "Five Determinisation Algorithms". In: *International Conference on Implementation and Application of Automata*. Springer. 2008, pages 161–170. DOI: 10.1007/978-3-540-70844-5_17.

[51] E. M. Gold. "Language Identification in the Limit". In: *Information and control*, volume 10, number 5, 1967, pages 447–474. DOI: 10.1016/S0019-9958(67)91165-5.

[52] M. Goldstein, D. Raz, and I. Segall. "Experience Report: Log-Based Behavioral Differencing". In: *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*. 2017, pages 282–293. DOI: 10.1109/ISSRE.2017.14.

[53] H. Gomaa. *Real-Time Software Design for Embedded Systems*. 1st edition. Cambridge University Press, 2016. ISBN: 978-1-139-64453-2.

[54] M. A. Gulzar, Y. Zhu, and X. Han. "Perception and Practices of Differential Testing". In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE. 2019, pages 71–80. DOI: 10.1109/ICSE-SEIP.2019.00016.

[55] D. Hendriks and K. Aslam. "A Systematic Approach for Interfacing Component-Based Software with an Active Automata Learning Tool". In: *Proceedings of the 11th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*. Springer. 2022, pages 216–236. DOI: 10.1007/978-3-031-19756-7_13.

[56] D. Hendriks, A. van der Meer, and W. Oortwijn. "A Multi-level Methodology for Behavioral Comparison of Software-Intensive Systems". In: *Proceedings of the 27th International Conference on Formal Methods for Industrial Critical Systems (FMICS)*. Springer International Publishing, 2022, pages 226–243. DOI: 10.1007/978-3-031-15008-1_15.

[57] D. Hendriks and W. Oortwijn. *Artifact for the paper 'gLTSdiff: A Generalized Framework for Structural Comparison of Software Behavior'*. 2023. DOI: 10.5281/zenodo.8096654.

[58] D. Hendriks and W. Oortwijn. "gLTSdiff: A Generalized Framework for Structural Comparison of Software Behavior". In: *Proceedings of the 26th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 2023, pages 285–295. DOI: 10.1109/MODELS58315.2023.00025.

[59] M. Heule and S. Verwer. “Software Model Synthesis using Satisfiability Solvers”. In: *Empirical Software Engineering*, volume 18, number 4, 2013, pages 825–856. DOI: 10.1007/s10664-012-9222-z.

[60] C. de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, 2010. ISBN: 978-0-521-76316-5.

[61] A. Hilbig. “Analysing Timing Behavior of Component-Based Software Systems”. Master’s thesis. University of Stuttgart, 2023. DOI: 10.18419/opus-13341.

[62] B. Hooimeijer. “Model Inference for Legacy Software in Component-Based Architectures”. Master’s thesis. Eindhoven University of Technology, 2020.

[63] 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)*. SciTePress, 2022, pages 146–158. DOI: 10.5220/0011145700003266.

[64] J. Hopcroft. “An n log n algorithm for minimizing states in a finite automaton”. In: *Theory of machines and computations*. Elsevier, 1971, pages 189–196. DOI: 10.1016/B978-0-12-417750-5.50022-1.

[65] J. Hopcroft, R. Motwani, and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. 3rd edition. Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 978-0-321-45536-9.

[66] F. Howar. “Active learning of interface programs”. PhD thesis. Technische Universität Dortmund, 2012. DOI: 10.17877/DE290R-4817.

[67] F. Howar, B. Jonsson, and F. Vaandrager. “Combining Black-Box and White-Box Techniques for Learning Register Automata”. In: *Computing and Software Science: State of the Art and Perspectives*. Springer International Publishing, 2019, pages 563–588. DOI: 10.1007/978-3-319-91908-9_26.

[68] F. Howar and B. Steffen. “Active Automata Learning in Practice: An Annotated Bibliography of the Years 2011 to 2016”. In: *Machine Learning for Dynamic Software Analysis: Potentials and Limits*. Springer, 2018, pages 123–148. DOI: 10.1007/978-3-319-96562-8_5.

[69] S.-M. Hsieh, C.-C. Hsu, and L.-F. Hsu. “Efficient Method to Perform Isomorphism Testing of Labeled Graphs”. In: *Computational Science and Its Applications (ICCSA)*. Springer Berlin Heidelberg, 2006, pages 422–431. DOI: 10.1007/11751649_46.

[70] M. Isberner, F. Howar, and B. Steffen. “The Open-Source LearnLib”. In: *Computer Aided Verification*. Springer International Publishing, 2015, pages 487–495. DOI: 10.1007/978-3-319-21690-4_32.

[71] M. Isberner, F. Howar, and B. Steffen. “The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning”. In: *Runtime Verification*. Springer International Publishing, 2014, pages 307–322. DOI: 10.1007/978-3-319-11164-3_26.

[72] E. Janssen. “Fingerprinting TLS Implementations Using Model Learning”. Master’s thesis. Radboud University, 2021.

[73] R. Jonk, J. Voeten, M. Geilen, T. Basten, and R. Schiffelers. *SMT-based verification of temporal properties for component-based software systems*. Technical report. ES Reports. Eindhoven University of Technology, 2020.

[74] 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*. Technical report ESR-2019-01. Eindhoven University of Technology, 2019.

[75] T. Kehrer, U. Kelter, P. Pietsch, and M. Schmidt. “Adaptability of Model Comparison Tools”. In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Association for Computing Machinery, 2012, pages 306–309. DOI: 10.1145/2351676.2351731.

[76] U. Kelter and M. Schmidt. “Comparing state machines”. In: *Proceedings of the 2008 international workshop on Comparison and versioning of software models*. 2008, pages 1–6. DOI: 10.1145/1370152.1370154.

[77] S. Klusener, A. Mooij, J. Ketema, and H. van Wezep. “Reducing Code Duplication by Identifying Fresh Domain Abstractions”. In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2018, pages 569–578. DOI: 10.1109/ICSME.2018.00020.

[78] J. Koskinen. *Software Maintenance Costs*. School of Computing, University of Eastern Finland, Joensuu, Finland, 2015, pages 1–3.

[79] I. Kurtev, J. Hooman, and M. Schuts. “Runtime Monitoring Based on Interface Specifications”. In: *ModelEd, TestEd, TrustEd: Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday*. Springer International Publishing, 2017, pages 335–356. DOI: 10.1007/978-3-319-68270-9_17.

[80] I. Kurtev, M. Schuts, J. Hooman, and D.-J. Swagerman. “Integrating Interface Modeling and Analysis in an Industrial Setting”. In: *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. SCITEPRESS, 2017, pages 345–352. DOI: 10.5220/0006133103450352.

[81] D. Kuske and A. Muscholl. “Communicating automata”. In: *Handbook of Automata Theory*. European Mathematical Society Publishing House, Zürich, Switzerland, 2021, pages 1147–1188. DOI: 10.4171/AUTOMATA-2/9.

[82] K. Lang, B. Pearlmutter, and R. Price. “Results of the Abbadingo One DFA Learning Competition and a New Evidence Driven State Merging Algorithm”. In: *International Colloquium on Grammatical Inference*. Springer. 1998, pages 1–12. DOI: 10.1007/BFb0054059.

[83] M. Leemans, W. van der Aalst, M. van den Brand, R. Schiffelers, and L. Lensink. “Software Process Analysis Methodology – A Methodology based on Lessons Learned in Embracing Legacy Software”. In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2018, pages 665–674. DOI: 10.1109/ICSME.2018.00076.

[84] M. Lehman and J. Ramil. "Software evolution in the age of component-based software engineering". In: *IEE Proceedings-Software*, volume 147, number 6, 2000, pages 249–255. DOI: 10.1049/ip-sen:20000922.

[85] M. Lehman. "Programs, Life Cycles, and Laws of Software Evolution". In: *Proceedings of the IEEE*, volume 68, number 9, 1980, pages 1060–1076. DOI: 10.1109/PROC.1980.11805.

[86] S. Lehnert. *A Review of Software Change Impact Analysis*. Technische Universität Ilmenau, 2011.

[87] H. Leung and L. White. "A Study of Integration Testing and Software Regression at the Integration Level". In: *Proceedings of the Conference on Software Maintenance*. IEEE. 1990, pages 290–301. DOI: 10.1109/ICSM.1990.131377.

[88] B. Li, X. Sun, H. Leung, and S. Zhang. "A survey of code-based change impact analysis techniques". In: *Software Testing, Verification and Reliability*, volume 23, number 8, 2013, pages 613–646. DOI: 10.1002/stvr.1475.

[89] S. Mahmood, R. Lai, and Y. S. Kim. "Survey of component-based software development". In: *IET software*, volume 1, number 2, 2007, pages 57–66. DOI: 10.1049/iet-sen:20060045.

[90] A. Mazurkiewicz. "Introduction to Trace Theory". In: *The Book of Traces*. World Scientific, 1995. Chapter 1, pages 3–41. DOI: 10.1142/2563.

[91] M. McIlroy, J. Buxton, P. Naur, and B. Randell. "Mass Produced Software Components". In: *Proceedings of the 1st international conference on software engineering*. 1968, pages 88–98.

[92] D. Menascé and V. Almeida. "Performance of Client/Server Systems". In: *Performance Evaluation: Origins and Directions*. Springer Berlin Heidelberg, 2000, pages 201–218. DOI: 10.1007/3-540-46506-5_8.

[93] T. Mens and T. Tourwé. "A Survey of Software Refactoring". In: *IEEE Transactions on Software Engineering*, volume 30, number 2, 2004, pages 126–139. DOI: 10.1109/TSE.2004.1265817.

[94] M. Merten, M. Isberner, F. Howar, B. Steffen, and T. Margaria. "Automated Learning Setups in Automata Learning". In: *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*. Springer Berlin Heidelberg, 2012, pages 591–607. DOI: 10.1007/978-3-642-34026-0_44.

[95] J. Midtgaard. "Control-Flow Analysis of Functional Programs". In: *ACM Computing Surveys*, volume 44, number 3, 2012, DOI: 10.1145/2187671.2187672.

[96] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., 1989. ISBN: 978-0-131-15007-2.

[97] A. Muscholl. "Analysis of Communicating Automata". In: *Language and Automata Theory and Applications (LATA)*. 2010, pages 50–57. DOI: 10.1007/978-3-642-13089-2_4.

[98] A. Nakamura, T. Saito, I. Takigawa, and M. Kudo. “Fast algorithms for finding a minimum repetition representation of strings and trees”. In: *Discrete Applied Mathematics*, volume 161, number 10-11, 2013, pages 1556–1575. DOI: 10.1016/j.dam.2012.12.013.

[99] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave. “Matching and Merging of Statecharts Specifications”. In: *29th International Conference on Software Engineering (ICSE)*. IEEE. 2007, pages 54–64. DOI: 10.1109/ICSE.2007.50.

[100] R. de Nicola and F. Vaandrager. “Three Logics for Branching Bisimulation”. In: *Journal of the ACM (JACM)*, volume 42, number 2, 1995, pages 458–487. DOI: 10.1145/201019.201032.

[101] T. Noergaard. *Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers*. Newnes, 2005. ISBN: 978-0-7506-7792-9.

[102] W. Oortwijn, D. Hendriks, A. van der Meer, and B. Huijbrechts. “Getting a Grip on the Ever-Changing Software in Cyber-Physical Systems”. In: *INSIGHT*, volume 25, number 4, 2022, pages 89–95. DOI: 10.1002/inst.12419.

[103] R. Paige and R. Tarjan. “Three Partition Refinement Algorithms”. In: *SIAM Journal on Computing*, volume 16, number 6, 1987, pages 973–989. DOI: 10.1137/0216062.

[104] F. Plasil and S. Visnovsky. “Behavior protocols for software components”. In: *IEEE transactions on Software Engineering*, volume 28, number 11, 2002, pages 1056–1076.

[105] J. Quante and R. Koschke. “Dynamic Protocol Recovery”. In: *14th Working Conference on Reverse Engineering (WCRE)*. IEEE. 2007, pages 219–228. DOI: 10.1109/WCRE.2007.24.

[106] V. Rajlich. “Software Evolution and Maintenance”. In: *Future of Software Engineering Proceedings*. Association for Computing Machinery, 2014, pages 133–144. DOI: 10.1145/2593882.2593893.

[107] G. Reger, H. Barringer, and D. Rydeheard. “Automata-based Pattern Mining from Imperfect Traces”. In: *ACM SIGSOFT Software Engineering Notes*, volume 40, number 1, 2015, pages 1–8. DOI: 10.1145/2693208.2693220.

[108] E. Rich. *Automata, Computability and Complexity: Theory and Applications*. 1st edition. Prentice-Hall, 2007. ISBN: 978-0-13-228806-4.

[109] J. de Ruiter and E. Poll. “Protocol State Fuzzing of TLS Implementations”. In: *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, 2015, pages 193–206. DOI: 10.5555/2831143.2831156.

[110] P. Runeson and M. Höst. “Guidelines for conducting and reporting case study research in software engineering”. In: *Empirical Software Engineering*, volume 14, number 2, 2009, pages 131–164. DOI: 10.1007/s10664-008-9102-8.

[111] M. Schuts, J. Hooman, and F. Vaandrager. "Refactoring of Legacy Software using Model Learning and Equivalence Checking: an Industrial Experience Report". In: *International Conference on Integrated Formal Methods*. Springer. 2016, pages 311–325. DOI: 10.1007/978-3-319-33693-0_20.

[112] E. Shihab, A. Hassan, B. Adams, and Z. M. Jiang. "An Industrial Study on the Risk of Software Changes". In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, 2012, pages 1–11. DOI: 10.1145/2393596.2393670.

[113] D. Shin, D. Bianculli, and L. Briand. "PRINS: scalable model inference for component-based system logs". In: *Empirical Software Engineering*, volume 27, 2022, pages 1–32. DOI: 10.1007/s10664-021-10111-4.

[114] J. Siegmund. "Program comprehension: Past, present, and future". In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Volume 5. IEEE. 2016, pages 13–20. DOI: 10.1109/SANER.2016.35.

[115] M. Sipser. *Introduction to the Theory of Computation*. 3rd edition. Cengage Learning, 2013. ISBN: 978-1-133-18779-0.

[116] M. Sokolova and G. Lapalme. "A systematic analysis of performance measures for classification tasks". In: *Information processing & management*, volume 45, number 4, 2009, pages 427–437. DOI: 10.1016/j.ipm.2009.03.002.

[117] O. Sokolsky, S. Kannan, and I. Lee. "Simulation-Based Graph Similarity". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer. 2006, pages 426–440. DOI: 10.1007/11691372_28.

[118] M.-A. Storey, N. Ernst, C. Williams, and E. Kalliamvakou. "The Who, What, How of Software Engineering Research: A Socio-Technical Framework". In: *Empirical Software Engineering*, volume 25, number 5, 2020, pages 4097–4129. DOI: 10.1007/s10664-020-09858-z.

[119] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. 2nd edition. Pearson Education, 2002. ISBN: 978-0-201-74572-6.

[120] TNO and Contributors to the GitHub community. *Model Inference and Differencing Suite (MIDS)*. Accessed: 2023-11-16. URL: https://github.com/TNO/MIDS.

[121] TNO and Contributors to the GitHub community. *Platform Performance Suite (PPS)*. Accessed: 2023-11-16. URL: https://github.com/TNO/PPS.

[122] K. Triantafyllidis, S. Niknam, and Y. Blankenstein. *PPS: From Methodology to Application in Industry*. Technical report TNO 2023 R12562. TNO, 2023. URL: https://repository.tno.nl/SingleDoc?docId=58556.

[123] F. Vaandrager. "Model Learning". In: *Communications of the ACM*, volume 60, number 2, 2017, pages 86–95. DOI: 10.1145/2967606.

[124] 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". In: *Journal of Systems and Software*, volume 111, 2016, pages 128–148. DOI: 10.1016/j.jss.2015.09.019.

[125] P. Vitharana. "Risks and Challenges of Component-based Software Development". In: *Communications of the ACM*, volume 46, number 8, 2003, pages 67–72. DOI: 10.1145/859670.859671.

[126] N. Walkinshaw and K. Bogdanov. "Automated Comparison of State-Based Software Models in terms of their Language and Structure". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)*, volume 22, number 2, 2013, pages 1–37. DOI: 10.1145/2430545.2430549.

[127] I. Warren. *The Renaissance of Legacy Systems: Method Support for Software-System Evolution*. Springer, 1999. ISBN: 978-1-85233-060-6.

[128] J. Whittle, J. Hutchinson, and M. Rouncefield. "The State of Practice in Model-Driven Engineering". In: *IEEE Software*, volume 31, number 3, 2014, pages 79–85. DOI: 10.1109/MS.2013.65.

[129] N. Yang. "Logs and Models in Engineering Complex Embedded Production Software Systems". PhD thesis. Eindhoven University of Technology, 2023.

[130] 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)*. 2019, pages 253–263. DOI: 10.1109/SANER.2019.8668007.

[131] N. Yang, P. Cuijpers, D. Hendriks, R. Schiffelers, J. Lukkien, and A. Serebrenik. "An interview study about the use of logs in embedded software engineering". In: *Empirical Software Engineering*, volume 28, number 2, 2023, DOI: 10.1007/s10664-022-10258-8.

[132] N. Yang, P. Cuijpers, R. Schiffelers, J. Lukkien, and A. Serebrenik. "An Interview Study of how Developers use Execution Logs in Embedded Software Engineering". In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE. 2021, pages 61–70. DOI: 10.1109/ICSE-SEIP52600.2021.00015.

[133] S. Yau, J. Collofello, and T. MacGregor. "Ripple effect analysis of software maintenance". In: *The IEEE Computer Society's Second International Computer Software and Applications Conference (COMPSAC)*. 1978, pages 60–65. DOI: 10.1109/CMPSAC.1978.810308.

[134] A. Zaidman, M. Pinzger, and A. van Deursen. "Software Evolution". In: *Encyclopedia of Software Engineering*. Taylor & Francis, 2010, pages 1127–1137.

Research data management

This thesis research has been carried out under the research data management policy of the Institute for Computing and Information Science of Radboud University, The Netherlands[2]. The following research datasets have been produced during this PhD research:

- Chapter 5:
  - Dennis Hendriks and Wytse Oortwijn, *Artifact for the paper 'gLTSdiff: A Generalized Framework for Structural Comparison of Software Behavior'*, Zenodo, 2023, DOI: `10.5281/zenodo.8096654`.

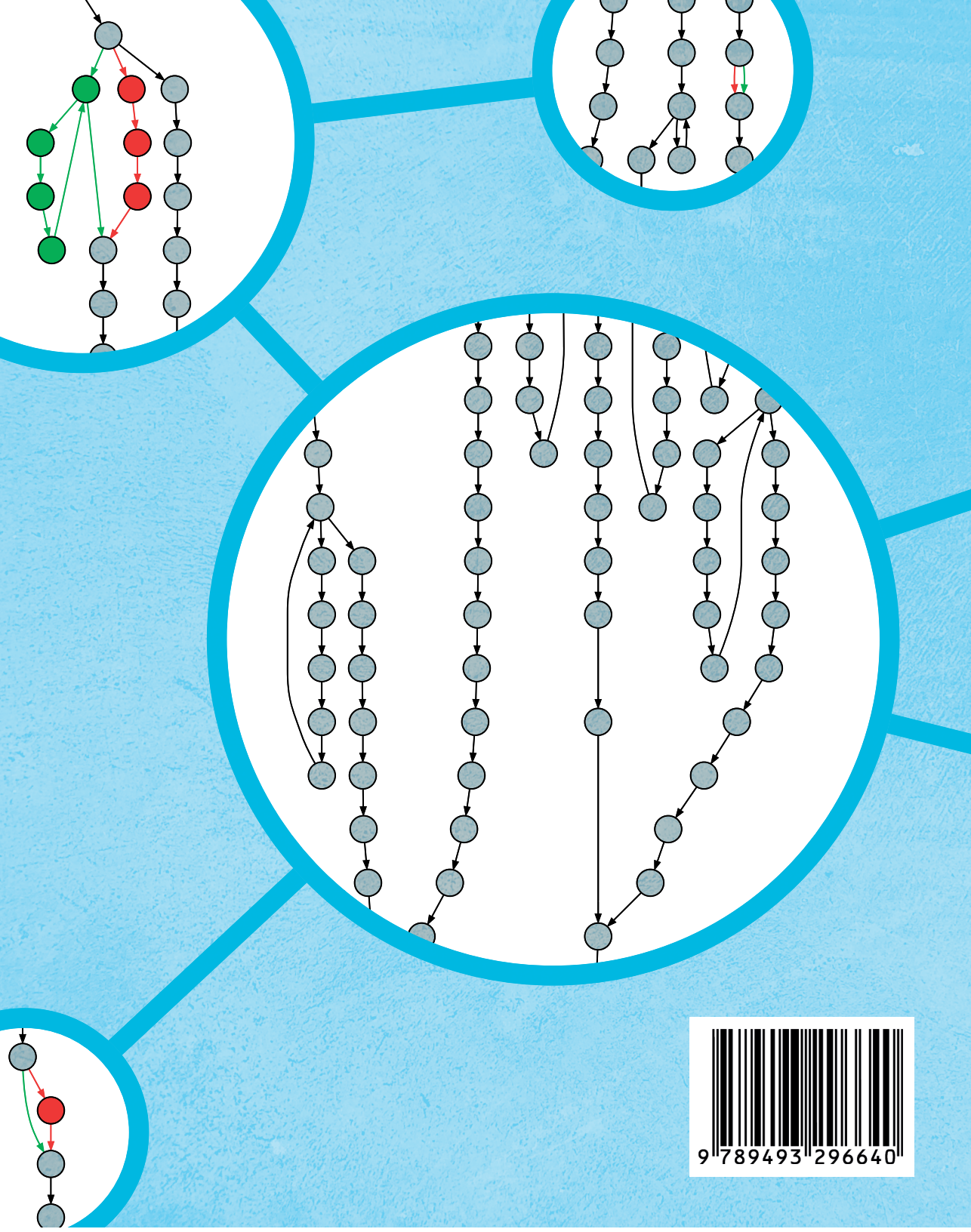
Besides this, the MIDS methodology is available as part of the open-source MIDS tool [120]. The materials for the remaining industrial case studies are not publicly available.

[2] `https://www.ru.nl/icis/research-data-management/`, last accessed November 17, 2023.

Curriculum vitae

Dennis Hendriks

1984	Born on July 15th in Nijmegen, The Netherlands.
1996 – 2002	Grammar school (Dutch: VWO / Gymnasium) at Merlet College, Cuijk, The Netherlands.
2002 – 2006	Bachelor Computer Science and Engineering (Dutch: Technische Informatica) at the Eindhoven University of Technology (TU/e), Eindhoven, The Netherlands.
2005 – 2007	Master Computer Science and Engineering at the Eindhoven University of Technology (TU/e), Eindhoven, The Netherlands.
2007 – 2016	Scientific programmer at the Mechanical Engineering department of the Eindhoven University of Technology (TU/e), Eindhoven, The Netherlands.
2015 (3 months)	Scientific programmer in a project at ASML, Veldhoven, The Netherlands.
2016 – 2020	Medior Research Fellow at the Embedded Systems Innovation (ESI) group of the Netherlands Organisation for Applied Scientific Research (TNO), Eindhoven, The Netherlands.
2020 – present	Senior Research Fellow at the Embedded Systems Innovation (ESI) group of the Netherlands Organisation for Applied Scientific Research (TNO), Eindhoven, The Netherlands.
2020 – present	Project lead of the Eclipse Supervisory Control Engineering Toolkit (Eclipse ESCET™) open-source project at the Eclipse Foundation.
2021 – present	Part-time researcher and PhD candidate at the Software Science group of the Institute for Computing and Information Sciences (iCIS) at the Radboud University, Nijmegen, The Netherlands.



9 789493 296640

Radboud Universiteit