Simplicity-Oriented Lifelong Learning of Web Applications

Dissertation

zur Erlangung des Grades eines

D o k t o r s d e r I n g e n i e u r w i s s e n s c h a f t e n

der Technischen Universität Dortmund

an der Fakultät für Informatik

von

Julius Alexander Bainczyk

Dortmund

2023

Tag der mündlichen Prüfung:
29. Januar 2024

Dekan:
Prof. Dr.-Ing. Gernot A. Fink

Gutachter:
Prof. Dr. Bernhard Steffen
Prof. Dr. Reiner Hähnle

Acknowledgements

This dissertation would not have been possible without all the remarkable people I have had the pleasure of meeting and working with along this journey. In the realm of research, every step is characterized by the influence of numerous hands and minds and I am deeply grateful for the impact each of them had on my work.

First and foremost, I extend my deepest gratitude to my supervisor, *Bernhard Steffen*, for giving me the opportunity to pursue my academic aspirations at his chair, and for his unwavering support, guidance, and encouragement throughout all those years. He fostered an environment where innovation and academic growth could flourish and where research was deeply aligned with the individual interests and passions of each person working in his department, providing a sense of fulfillment in academic work.

I would also like to express my sincere gratitude to *Rainer Hähnle* for his role as my co-supervisor, to *Ben Hermann* for serving as the chair of my dissertation committee, and to *Jakob Rehof* for his contribution as a member of my committee. Together with *Bernhard Steffen*, they made my doctoral defense a once-in-a-lifetime experience.

In one way or another, every dissertation is the result of collective effort. For this reason, I would like to thank all my colleagues that I had the privilege to work with, namely *Steve Boßelmann*, *Daniel Busch*, *Johannes Düsing*, *Markus Frohme*, *Anemone Kampkötter*, *Dawid Kopetzki*, *Michael Lybecait*, *Alnis Murtovi*, *Stefan Naujokat*, *Gerrit Nolte*, *Oliver Rüthing*, *Maximilian Schlüter*, *Steven Smyth*, *Jonas Schürmann*, *Tim Tegeler*, *Dominic Wirkner*, and *Philip Zweihoff*. Thank you for working with me on all the projects and publications, for the little chats in the coffee kitchen, for the memories we created inside and outside of the chair, and for celebrating the ups and dealing with the downs that our profession entails.

In addition to that, I would also like to express my genuine appreciation to *Julia Rehder*, who I consider to be a cornerstone of the chair. Not only did she make sure that we never saw the bottom of the chocolate drawer, she also always had a listening ear for anyone who needed it.

Last but not least, I would like to thank *Annika Fuhge* for her invaluable support throughout my academic career. Especially in the days, weeks and months I spent with my head above my desk writing this dissertation, her emotional support has been a beacon of strength that has guided me through moments of doubt and uncertainty.

Abstract

Nowadays, web applications are ubiquitous. Entire business models revolve around making their services available over the Internet, anytime, anywhere in the world. Due to today's rapid development practices, software changes are released faster than ever before, creating the risk of losing control over the quality of the delivered products. To counter this, appropriate testing methodologies must be deeply integrated into each phase of the development cycle to identify potential defects as early as possible and to ensure that the product operates as expected in production. The use of low- and no-code tools and code generation technologies can drastically reduce the implementation effort by using well-tailored (graphical) Domain-Specific Languages (DSLs) to focus on what is important: the product. DSLs and corresponding Integrated Modeling Environments (IMEs) are a key enabler for quality control because many system properties can already be verified at a pre-product level. However, to verify that the product fulfills given functional requirements at runtime, end-to-end testing is still a necessity.

This dissertation describes the implementation of a *lifelong learning* framework for the continuous quality control of web applications. In this framework, models representing user-level behavior are mined from running systems using active automata learning, and system properties are verified using model checking. All this is achieved in a continuous and fully automated manner. Code changes trigger testing, learning, and verification processes which generate feedback that can be used for model refinement or product improvement. The main focus of this framework is simplicity. On the one hand, it allows Quality Assurance (QA) engineers to apply learning-based testing techniques to web applications with minimal effort, even without writing code; on the other hand, it allows automation engineers to easily implement these techniques in modern development workflows driven by Continuous Integration and Continuous Deployment (CI/CD).

The effectiveness of this framework is leveraged by the Language-Driven Engineering (LDE) approach to web development. Key to this is the text-based DSL iHTML, which enables the instrumentation of user interfaces to make web applications *learnable by design*, i.e., they adhere to practices that allow fully automated inference of behavioral models without prior specification of an input alphabet. By designing code generators to generate instrumented web-based products, the effort for quality control in the LDE ecosystem is minimized and reduced to formulating runtime properties in temporal logic and verifying them against learned models.

Attached Publications

The publications covered by this dissertation are listed below. They were written and published in collaboration with other authors. A summary of each publication, including my contributions, is given in Section 1.1.

I Alexander Bainczyk, Alexander Schieweck, Bernhard Steffen, and Falk Howar. **Model-Based Testing Without Models: The TodoMVC Case Study**. In: *ModelEd, TestEd, TrustEd: Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday*. Cham: Springer International Publishing, 2017. DOI: 10.1007/978-3-319-68270-9_7

II Alexander Bainczyk, Bernhard Steffen, and Falk Howar. **Lifelong Learning of Reactive Systems in Practice**. In: *The Logic of Software. A Tasting Menu of Formal Methods: Essays Dedicated to Reiner Hähnle on the Occasion of His 60th Birthday*. Cham: Springer International Publishing, 2022. DOI: 10.1007/978-3-031-08166-8_3

III Tim Tegeler, Sebastian Teumert, Jonas Schürmann, Alexander Bainczyk, Daniel Busch, and Bernhard Steffen. **An Introduction to Graphical Modeling of CI/CD Workflows with Rig**. In: *Leveraging Applications of Formal Methods, Verification and Validation*. Cham: Springer International Publishing, 2021. DOI: 10.1007/978-3-030-89159-6_1

IV Alexander Bainczyk, Steve Boßelmann, Marvin Krause, Marco Krumrey, Dominic Wirkner, and Bernhard Steffen. **Towards Continuous Quality Control in the Context of Language-Driven Engineering**. In: *Leveraging Applications of Formal Methods, Verification and Validation. Software Engineering*. Cham: Springer Nature Switzerland, 2022. DOI: 10.1007/978-3-031-19756-7_22

V Alexander Bainczyk, Daniel Busch, Marco Krumrey, Daniel Sami Mitwalli, Jonas Schürmann, Joel Tagoukeng Dongmo, and Bernhard Steffen. **CINCO Cloud: A Holistic Approach for Web-Based Language-Driven Engineering**. In: *Leveraging Applications of Formal Methods, Verification and Validation. Software Engineering*. Cham: Springer Nature Switzerland, 2022. DOI: 10.1007/978-3-031-19756-7_23

VI Daniel Busch, Gerrit Nolte, Alexander Bainczyk, and Bernhard Steffen. **ChatGPT in the Loop: A Natural Language Extension for Domain-Specific Modeling Languages**. In: *Bridging the Gap Between AI and Reality*. Cham: Springer Nature Switzerland, 2024. DOI: 10.1007/978-3-031-46002-9_24

VII Daniel Busch, Alexander Bainczyk, and Bernhard Steffen. **Towards LLM-Based System Migration in Language-Driven Engineering**. In: *Engineering of Computer-Based Systems*. Cham: Springer Nature Switzerland, 2024. DOI: 10.1007/978-3-031-49252-5_14

Contents

Chapter 1

Introduction

Web applications have become an indispensable part of our daily lives. Organizations whose business models depend on their online services must ensure that these services perform as expected from the user's perspective. End-to-end testing is a method of verifying the user-level behavior of web applications by posing tests against a running instance through its browser interface. In doing so, much of an application's business logic can be covered, as interactions with the web interface trigger the execution of server-side code that may interact with a database or even other web-based services, and ultimately send data back to the web browser. This testing approach becomes particularly effective in the context of test automation, as much of the manual testing effort can be reduced or even eliminated. As more development teams adopt agile practices and systems are deployed more frequently, even multiple times a day, test automation enables organizations to respond quickly to changing business needs and customer demands while maintaining control over system quality.

To ensure product quality in such fast-paced environments, test automation is essential. However, the effort required to develop and maintain automated tests is often underestimated due to their complexity [27, 85]. An approach where tests are not only executed but also generated automatically is *Active Automata Learning* [67, 88, 95], which has proven to be a suitable method for end-to-end testing of web applications, as several examples have already demonstrated [12, 67, 80, 88, 89]. In active automata learning, behavioral models are inferred from running systems by executing carefully constructed test queries over some input alphabet. The inferred models are then used to verify system properties using model checking via temporal logic [86]. Several publications [16, 78, 108] have already proposed automata learning for the continuous quality control of web applications in modern development workflows. Central to these *lifelong learning*, sometimes also called *never-stop learning*, approaches is the automated model inference, the verification of system properties via model checking, and the observation of the running system for runtime verification. Frohme et al. already demonstrated a lifelong learning scenario for context-free systems in [37] and illustrated the implementation of a corresponding runtime monitor for Systems of Procedural Automata (SPA) [36] in [35]. However, these practices have been evaluated on abstract models and have not yet been applied to real systems.

To apply these continuous automata learning practices to real-world applications, code-centric frameworks such as LearnLib [47, 58, 69], libalf [17, 60] and Tomte [3, 101], of which LearnLib is the only one actively maintained, have been developed. While these frameworks mostly target code-affine developers, tools such as LearnLib Studio [15, 67], LBTest [66], and Webtest [87] aim to lower the entry barrier for a non-expert audience by providing visual guidance for modeling learning setups via dedicated graphical user interfaces. However, these tools are either too general and require extensions depending on the target system, are difficult to integrate with existing development stacks and therefore make it difficult to automate processes, or suffer from both. This motivated the development of our tool ALEX [6, 11], which takes a simplicity-oriented approach to learning web

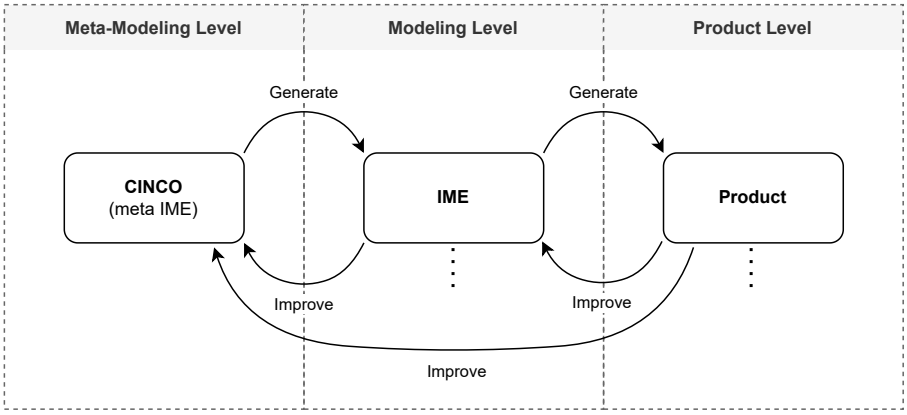


Figure 1.1: The continuous improvement cycle in the LDE ecosystem.

applications by allowing users to model learning setups in a low-code manner in the web browser.

With regard to low-code tool development, Steffen et al. present the Language-Driven Engineering (LDE) approach [94], which includes product stakeholders in the development process by using (graphical) Domain-Specific Languages (DSLs) [33]. Central to LDE is the *CINCO SCCE Meta Tooling Framework* [74], also referred to as CINCO, which has already been used in several research and industrial contexts [22, 74, 75, 99, 109] and which enables language engineers to develop graph-based DSLs and to generate corresponding Integrated Modeling Environments (IMEs) from them. In this work, the landscape around CINCO, its IMEs and generated products is called the *LDE ecosystem* [18]. As illustrated in Figure 1.1, applications in the ecosystem form a *meta-level hierarchy*, with CINCO at the *meta-modeling level*, CINCO-based IMEs at the *modeling level*, and IME products at the *product level*. Development in the LDE ecosystem follows a continuous improvement cycle: Feedback received at any level is propagated up the meta-level hierarchy, where the issue is addressed and the tool is improved. From there, changes are propagated down the hierarchy, followed by migration and testing processes at each level until the product is reached. Although low-code tools and DSLs are generally said to improve software quality [53], quality control of web-based products developed using the LDE approach still poses challenges to language designers, modelers and quality assurance experts. The main reason for this is the generative approach of LDE. Code generators, both for IMEs and IME products, are written manually and are therefore susceptible to human error, which can result in code that cannot be compiled in the first place or in application behavior that deviates from the semantics of the underlying language. As a consequence, the testing of generated applications is still necessary.

This dissertation aims to simplify the quality assurance of web applications by combining the benefits of the LDE approach with the capabilities of active automata learning. Key to this is a framework for learning-based continuous quality control of web applications, which is designed for

- *Simplicity* – Lifelong learning of web applications is made accessible to non-expert audiences by providing web-based low-code tools designed for ease of use.

- *Control* – Improved control over the evolution of web applications developed with the LDE approach by exploiting the benefits of DSLs and the generative approach to generate applications that are *learnable by design*.

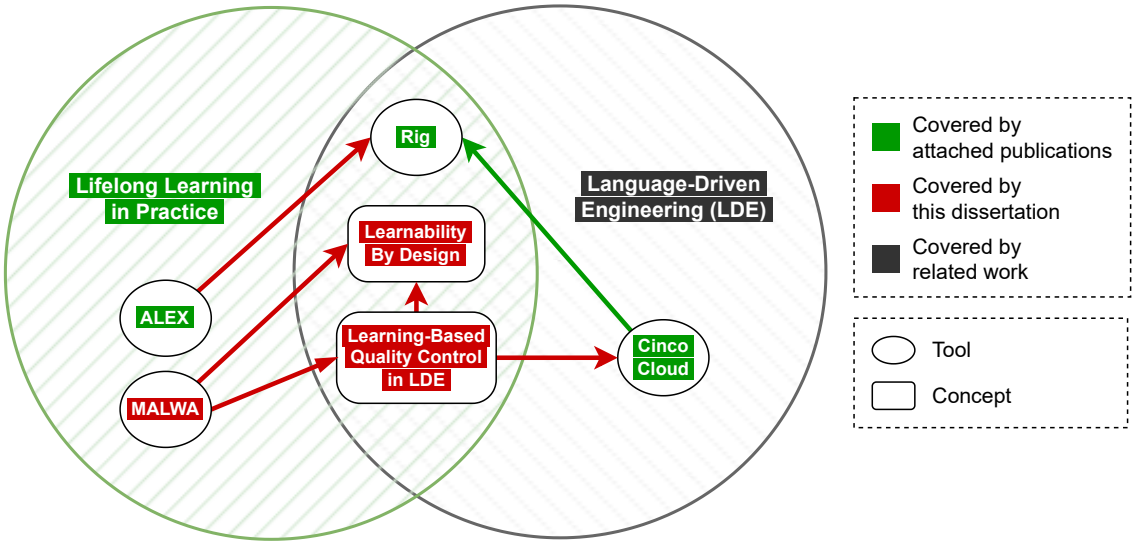


Figure 1.2: Contributions of this dissertation.

- *Automation* – Lifelong learning practices are integrated into Continuous Integration and Continuous Deployment (CI/CD) providers. As a result, learning and verification processes can be fully automated, which allows developers to easily adopt this technology in their development processes.

1.1 My Contributions

The motivation to simplify the quality assurance of web applications via automated testing arose from my experience as a student in the lecture "Webtechnologies 2" at the TU Dortmund University. There, students are taught modern web technologies and tasked to develop their own web application during the semester. Part of this process is the verification of runtime system properties by means of analyzing inferred automaton models obtained from active automata learning, which a majority of peers struggled with. The main reason for this was that existing tools and frameworks were not accessible enough for a non-expert audience, as they were designed to be general purpose which increased complexity and therefore hindered their adaptation. As a consequence, Alexander Schieweck and I developed ALEX [6, 11], an open-source, web-based tool that allows users to apply practices from automata learning to the domain of web applications in a low-code manner. Since then, I have gained experience in this field from being a recurring guest lecturer in "Webtechnologien 2" where ALEX has gradually improved the students' results over the past five years, but also from being a supervisor of student project groups in the context of web development and from my participation in industrial cooperations. I could further apply this knowledge to the LDE ecosystem which includes the meta-modeling tool CINCO and especially the CINCO product DIME [19], our IME for the model-driven development of full-stack web applications. In this context, I focused on ensuring the quality of the DIME code generator as well as of web applications generated from DIME by applying traditional end-to-end tests and learning-based testing practices on DIME products.

Figure 1.2 provides a diagram with my main contributions to the field of active automata learning and to the LDE ecosystem. All in all, I consider my contributions to be threefold, covering the following topics.

A simplicity-oriented approach to lifelong learning of web applications The first contribution deals with the development of a framework for the continuous quality control of web applications using active automata learning. This *lifelong learning* framework has been designed with a focus on simplicity [64]. It allows non-experts to infer and then verify formal models of web applications and provides all means to control their evolution in a low-code fashion. To be precise, this dissertation concentrates on the extensions of the ALEX tool beyond its capabilities presented in previous work [11, 12]. These extensions allow users

- to generate redundancy free test-suites via model-based test generation from discrimination trees [52] used by the TTT algorithm [48],
- to use predefined system tests for a targeted counterexample search which guarantees a certain quality of learned models by establishing a common level of abstraction for modeling system tests and input alphabets, and
- to visualize and analyze changes between software versions by means of *difference trees*, i.e., tree-based representations of the differences between generated regression test suites and *difference automata*, i.e., automaton models that include all sequences indicating behavioral differences between two systems.

Moreover, I provided the means to allow the tool to be deeply integrated in modern development stacks that are driven by CI/CD technologies to fully automate learning, verification and refinement processes.

The concept of learnability-by-design in the context of web applications The second contribution addresses one of the major pain points when creating learning setups, namely the assembly of an adequate input alphabet and the manual implementation of system-specific execution logic for system inputs. However, the challenge is not only the implementation of execution logic, but also its continuous manual adaption to system changes. In the domain of web applications, changes to test execution logic can only be verified after the updated system has been deployed, e.g., to a test environment. This can cause delays in the development workflow in fast-paced environments, or result in insufficient testing as the application and its tests diverge more and more with each change. To mitigate these issues, this dissertation illustrates the concept of *learnability-by-design*, which consists of the following aspects:

- The DSL iHTML that developers can adapt to instrument Hypertext Markup Language (HTML) code with information relevant for testing and learning. First, this includes instructions for user-level interactions with specific elements on a website, such as clicks and form submissions. Second, this concerns data handling, i.e., by providing a notation to load test data from external resources during a learning process, e.g., for filling forms with test data, and by extracting data from the Document Object Model (DOM) of the website. Third and finally, this includes the explicit identification of *quiescent states* [102], i.e., states in which the system is at rest and waits for external input to identify when subsequent actions can be performed during testing.
- A learning setup where, starting with an initially empty input alphabet, symbols are mined on-thy-fly just by interacting with the target website through a learning algorithm. When the algorithm visits the website, instrumented parts of the DOM

are interpreted and translated into concrete input symbols. Consequently, web applications can be learned fully automatically, without explicit specification of an input alphabet.

- The tool Malwa, which gives users access to such a learning setup via a web-based user interface that only requests users to enter the Uniform Resource Locator (URL) to an instrumented application. Learned models reflect interactions with the user interface of the target application, presented in a way that simplifies visual analysis.

Practices for learning-based quality control in language-driven engineering The third and final contribution focuses on my developments towards a holistic approach to quality control in the LDE ecosystem. Keys to this are automation and the integration of learning-based testing as a measure to ensure behavioral correctness of generated applications. By lifting LDE to the web with CINCO Cloud, the provision of the meta IME and the generation and delivery of IMEs is completely automated. This does not only eliminate any system-specific dependencies, which are required when working with an Eclipse-based stack, but also simplifies the propagation of changes from the meta-level to the modeling level: domain experts always have an up-to-date IME. With CINCO Cloud [10], we control the language development, with the DSL Rig [98], we control build and delivery processes of products and with the tooling around ALEX [11] and Malwa, we ensure that the web-based products fulfill given functional requirements. I demonstrate how the learnability-by-design approach can be integrated in a web-centric LDE ecosystem, where code generators can be designed so that generated products become automatically learnable. This allows language engineers to verify code generators by generating a battery of test applications that are learnable by design and simplifies the quality assurance effort for domain experts that model web applications.

1.2 Context of Attached Publications

As a part of this dissertation, several concepts, tools, frameworks and practices have been developed and demonstrated that aim to simplify the quality control of web applications via learning-based testing by exploiting domain-specificity. A demonstration of the simplicity of using ALEX for learning web applications is presented in [12] and [13] illustrates the lifelong learning framework. Then, publication [98] focuses on the benefits of domain-specificity on the example of visual CI/CD pipeline modeling. Publication [9] discusses the benefits of the LDE ecosystem for quality control as a whole, outlines the relevance of the lifelong learning framework in this context and sketches the idea of learnability-by-design. This is followed by the publication [10] which talks about lifting the LDE toolchain to a fully web-based, holistic development environment with CINCO Cloud. Finally, the last publication [21] highlights the benefits of learnability-by-design in LDE for the verification of web applications generated by combining graphical DSLs and large language models.

Model-Based Testing Without Models: The TodoMVC Case Study

This paper [12] demonstrates a case study highlighting the simplicity of ALEX [6, 11] regarding its ability to create and compare learning experiments. For this purpose, the behavioral conformance of a web application implemented with different technology stacks was verified. Subject of the study was the TodoMVC project [5], which includes over 70 implementations of the same web-based and client-side task management system,

each developed with a different JavaScript framework. Key to the comparison was the establishment of a common behavioral language based on a given textual specification. During the study, input alphabets for 27 TodoMVC variants have been implemented systematically in ALEX with a focus on reuse of execution logic. By manually analyzing the DOM of the HTML documents of each implementation, structural similarities could be extracted and translated into reusable execution logic, which benefited the adaptation of the input alphabet to other implementations. Finally, the selected implementations have been learned and compared on a behavioral level to detect discrepancies.

My contribution: *The presented concepts were discussed among all authors. I co-authored all sections. The planing and the execution of the case study were conducted by Alexander Schieweck and myself in equal parts.*

Lifelong Learning of Reactive Systems in Practice

This paper [13] presents our work resulting from the DFG project *"Constraint-Based Operational Consistency of Evolving Software Systems (COCoS)"* whose initial results were documented by Reiner Hähnle in [41]. In the paper, a framework for controlling the evolution of software via learning-based testing is presented. Key to this *lifelong learning* framework is a continuous improvement cycle where behavioral models of black-box systems are inferred via model learning, semantic system properties are verified via model checking, and the system is observed at runtime. In addition, this cycle emphasizes the processes that occur between software iterations for controlling the evolution. Minimal regression test suites generated from internal data structures of the TTT algorithm [48] are used to learn new models more efficiently in successive iterations. Moreover, changes can be controlled more easily by automatically generating *difference trees* and *difference automata*, which allow the visualization of behavioral changes as well the appliance of formal verification techniques. At last, the publication demonstrates the capabilities of the cycle on the example of a simulation software for an adaptive cruise control system that has been implemented by students in yearlong group project. As a part of this paper, the simulator has been extended to allow its control via an HTTP-based interface in order to learn it with the ALEX tool.

My contribution: *The presented concepts were discussed among all authors. I co-authored all sections, planned and executed the case study and I am main author of sections 3 – 6. Further, I also implemented the HTTP interface for the simulator and the necessary extensions for the ALEX tool that enable users to apply lifelong learning practices.*

An Introduction to Graphical Modeling of CI/CD Workflows with Rig

The paper [98] uses the example of authoring CI/CD pipelines to show how typical software development tasks can be controlled and simplified by exploiting domain specificity. Focus of the publication is Rig [100], a CINCO-based IME that allows users to model CI/CD pipelines in a graphical modeling environment. The tool is presented as a prime example illustrating the benefits for the quality control of software artifacts through the integration of graphical DSLs and appropriate tool support into software development processes. Visualization bridges the communication gap between domain experts and non-professionals, language constraints, both syntactic and semantic, make it impossible to create invalid models and full code generation ensures that generated code is always executable. For demonstration purposes, a CI/CD pipeline is created for an implementation of TodoMVC

that builds, tests, packages and deploys the application to a static file hosting provider.

My contribution: *The presented concepts were discussed among all authors. I am main author of section 3. The implementation of Rig was primary performed by Sebastian Teumert under the supervision of Tim Tegeler.*

Towards Continuous Quality Control in the Context of Language-Driven Engineering

This paper [9] discusses the impact of domain-specificity on quality assurance in the LDE ecosystem around CINCO, DIME and DIME products, where static and dynamic validation methods form the cornerstone to ensure quality. While static validation concerns language constraints and their implementation in modeling environments, dynamic validation aims to verify system properties by testing the product. The paper demonstrates these measures on the development of EquninOCS, an editorial system that has been implemented in DIME. In terms of dynamic validation, the publication describes the integration of lifelong learning practices introduced in [13] in the LDE context. Key to this is automation: It is described how ALEX integrates with common CI/CD vendors so that properties of generated products are verified automatically via learning-based testing each time code is changed. Moreover, the publication illustrates a continuous feedback and improvement loop in the LDE ecosystem consisting of *path-up* and *tree-down* effects: On each level, feedback can occur that triggers changes on the same level or on a level above which then needs to be propagated down the meta-level hierarchy. Finally, the paper sketches *learnability-by-design* practices which leverage the LDE toolchain to further reduce manual quality assurance effort. This includes the generation of appropriate test artifacts from graphical models and the instrumentation of generated application code to enable simpler or even automated testing, learning, and runtime verification.

My contribution: *The presented concepts were discussed among all authors. I co-authored all sections and I am main author of section 2, 3 and 5. Further, I implemented the CI/CD pipeline using Rig and developed the necessary extensions for ALEX with the help of Marco Krumrey.*

CINCO Cloud: A Holistic Approach for Web-Based Language-Driven Engineering

Paper [10] is about the demonstration of CINCO Cloud, a web-based development platform which fully lifts the LDE toolchain, i.e., modeling as well as meta-modeling environments to the web. The tool is an implementation of the concepts presented in [113] and a further development of the prototype from the subsequent PhD thesis [111]. CINCO Cloud pursues the same objectives as Web-GME [65, 106], Sirius Web [29, 103], and emf.cloud [28], but focuses on a coordinated and purely web-based approach to LDE. This covers the design of graph-based languages, the generation and provision of corresponding IMEs and the deployment of products generated from graphical models. The paper therefore focuses on the presentation of a scalable and service-oriented system architecture that enables a highly automated development workflow for LDE. CINCO Cloud is demonstrated on the example of *WebStory*, a DSL for building web-based Point-and-Click adventures introduced in previous research [40, 56], from language design to the deployment of a modeled adventure to the web.

My contribution: *The presented concepts and technologies were discussed among all authors. I co-authored all sections except section 2 and section 6.1 and I am main author of section 3. I also implemented the system architecture of CINCO Cloud with the help*

of Marco Krumrey, while Daniel Sami Mitwalli and Joel Tagoukeng Dongmo worked on web-based (meta-)modeling environments under my supervision.

ChatGPT in the Loop – A Natural Language Extension for Domain-Specific Modeling Languages

This paper [21] is about exploring the benefits of large language models for code generation in the context of LDE. It introduces the idea of a two-step generation process where code generated from DSLs contains intersection points for code generated from natural language. The approach is illustrated using the *WebStory* language: While users model a basic application frame using the DSL, the game logic is expressed using natural language. Due to the nature of generative artificial intelligence, generated applications must be tested to verify that they exhibit the desired runtime behavior. For this purpose, *learnability-by-design* practices are applied by implementing code generators in such a way that they generate instrumented web applications. As a result, behavioral models can be inferred fully automatically for each WebStory product using active automata learning and validated using model checking.

My contribution: *The presented concepts and technologies were discussed among all authors. I am main author of section 2.3, section 3.4 and section 4.5 and co-authored section 1 and section 6. I provided a code frame for instrumenting arbitrary WebStory products, while Daniel Busch and Gerrit Nolte embedded this knowledge in the DSL-based code generator.*

Towards LLM-based System Migration in Language-Driven Engineering

This paper [20], which won the Best Short Paper Award at the *8th International Conference on the Engineering of Computer Based Systems ECBS 2023*, deals with semi-automated system migration from a source language to a target language using Large-Language Models (LLM). It builds on the ideas presented in the previous paper [21] and discusses a divide-and-conquer approach that decomposes a given problem domain, in this case code migration, into tasks that are just small enough for an LLM to handle reliably. The approach is demonstrated in the context of language-driven engineering using the *WebStory* DSL, where the code generator and the *prompt frame*, which translates natural language descriptions into game logic (see [21]), are migrated using an LLM to eventually generate TypeScript code instead of JavaScript code. To verify that the runtime behavior of generated applications after the system migration remains unchanged, the practices for learning-based testing and evolution control described in [13] are applied. In order to further simplify this process, code generators are designed to generate instrumented web applications that are learnable by design due to the instrumentation language presented in this dissertation. The paper shows that system properties can be verified on learned models using model checking, and that potential differences can be inferred and analyzed using *difference automata*, see Section 3.5.1. Finally, traces of divergent behavior are passed to the domain expert, who refines the LLM-based system migrator.

My contribution: *The presented concepts and technologies were discussed among all authors. I am main author of section 2.2 and co-authored section 1, section 3 and section 4. I provided the instrumentation language for instrumenting WebStory products and performed the verification of the system migration using a setup to infer and visualize differences between learned models, while Daniel Busch engineered the prompts to automatically migrate*

code generators and prompt frames.

1.3 Organization of this Dissertation

This cumulative dissertation is organized as follows: After introducing my contributions and the context of the attached publications in this chapter, Chapter 2 provides the necessary preliminaries related to black-box testing of web applications using active automated learning, the principles of language-driven engineering, and existing tools relevant to this work. Chapter 3 then illustrates the different facets of the framework for lifelong learning of web applications. After that, the concept of learnability-by-design is outlined and demonstrated on a practical example in Chapter 4. The measures for product-level quality control in a web-centric LDE ecosystem using active automata learning are the subject of Chapter 5. Finally, this dissertation ends with a conclusion and an outlook on future research in Chapter 6.

Chapter 2

Background

This chapter covers the foundations and tools used to implement the proposed Language-Driven Engineering (LDE) supported lifelong learning cycle. Therefore, an introduction to the structure of web applications and common end-to-end testing methods is given, followed by a presentation of the principles of automata learning and an overview of the LDE approach. In this context, the tools of relevance in this work, namely ALEX and CINCO are briefly introduced.

2.1 Web Applications

Web applications follow the client-server model [97] and can be divided into three logical layers, sometimes called tiers. The first is the *presentation layer*, which represents the user interface rendered in a web browser. The second layer is called the *business layer*, which contains the business logic of the application responsible for processing data. Finally, the *data layer* takes care of the persistence and management of the data, e.g. on the basis of a database system. In the further course of this work, the presentation layer will be referred to as the *frontend* and the last two layers will be combined as the *backend* of a web application.

In "traditional" web applications, Hypertext Markup Language (HTML) documents are pre-rendered in the backend and sent to the client. For each request, the website is completely reloaded and rebuilt in the browser. This stands in contrast to single-page applications, where all necessary static resources are loaded with an initial request, and additional data is requested asynchronously. With this method, only partial updates are made dynamically using JavaScript instead of a full-page reload. Typically, full-page reloads are also triggered when the Uniform Resource Locator (URL) changes. However, single-page applications employ client-side scripts that can manipulate the URL without full-page reloads and react to URL changes by updating parts of the document. The process is illustrated in Figure 2.1: First, the client sends an Hypertext Transfer Protocol (HTTP) request to the backend through a public Application Programming Interface (API)❶, which can be designed according to the Representational State Transfer (REST) principles [30]. The request is then processed, which may involve creating, loading, modifying, or deleting data in a database❷. The server then responds to the request❸, and if data is transferred, it is done using a data exchange format, which in modern web applications is typically JSON. Finally, upon receiving the response, the HTML document is dynamically updated using JavaScript to represent the changes ❹. In single-page applications, the HTML document often reflects the state of the application, which is anchored in the underlying database. Nonetheless, nuances such as client-only states imply that this reflection is not always a direct representation of the persisted data.

In this dissertation, the emphasis is put on frontends which are typically developed with HTML, Cascading Style Sheets (CSS) and JavaScript or other higher level languages that compile to these technologies. Web browsers display HTML documents that define a

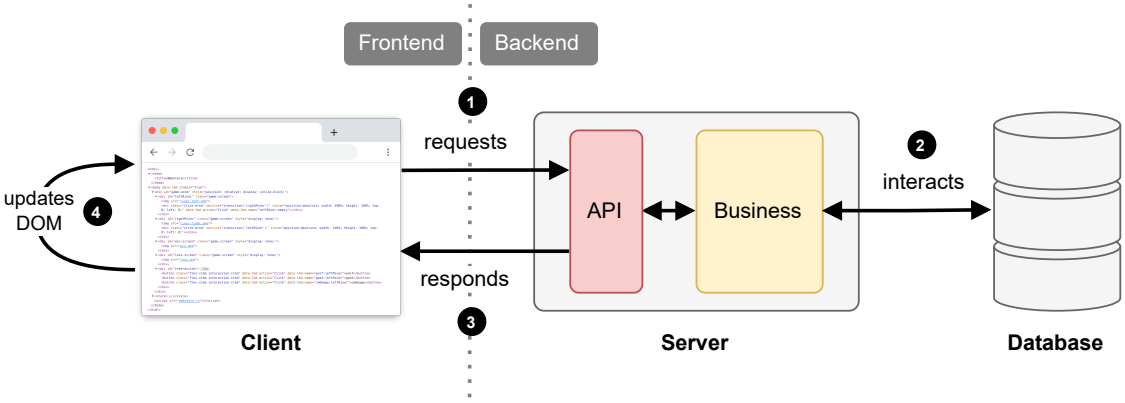


Figure 2.1: Three tier architecture of single-page applications.

website's structure. HTML is a language similar to XML in which elements are identified using tags. These tags can have attributes and can be nested to create a parent-child relationship. In the following example

```
<div class="bordered">Hello!</div>
```

`<div></div>` is the tag and `class="bordered"` is a key-value pair where the attribute `class` is assigned the value `bordered`. Each browser allows developers to modify the HTML document using the Document Object Model (DOM) [104] API with JavaScript. The DOM represents the document as a rooted tree containing element nodes which refer to the tags and attribute and text nodes that are children of element nodes. Using this API, new elements can be created programmatically and existing elements can be removed or rearranged within the tree. For example, single-page applications make extensive use of the DOM by composing the HTML document directly in the browser using JavaScript, rather than pre-computing it on the server and delivering it to the client.

CSS is a Domain-Specific Language (DSL) for defining the appearance of element nodes in the DOM. The means to do this is by specifying CSS *rules*, which consist of a set of styling properties and a *selector* that indicates to which elements the properties are applied to. For example, in the rule

```
.bordered { border:  2px solid red; }
```

`.bordered` denotes the selector and it tells the browser to draw a red border with a width of two pixels around each element node that has the value `bordered` in the `class` attribute, such as the node from the example above. In addition, CSS selectors can also be used to access element nodes in the DOM with JavaScript. The expression

```
document.querySelectorAll(".bordered");
```

returns a list of all element nodes that have the value `bordered` in their `class` attribute. However, this only poses a small example for how elements can be accessed and a complete listing of all selectors can be found in [105]. Choosing selectors for elements becomes particularly relevant in the context of frontend testing, as the following section explains.

2.2 End-to-End Testing

Software testing methods can be represented as what is commonly called the *testing pyramid*. At the bottom are tests that are close to the application code, such as unit tests. As one

moves from the bottom to the top of the pyramid, there are fewer references to actual code, but also more expensive tests to implement and maintain, and the longer it takes to get feedback from those tests. At the top of the pyramid are typically end-to-end tests, which fall into the category of black-box testing, since tests are executed against a running instance of the system without making any assumptions about its inner workings. In the context of web applications, tests are posed either against the provided HTTP-based APIs or, as considered in this work, against the browser interface to test user-level behavior. Fundamental to the implementation of such tests are browser automation frameworks such as Selenium [92], which can be used to programmatically simulate user input by interacting with the browser through the DOM API. Developers implementing automated end-to-end tests typically face the following challenges:

Test flakiness Test flakiness describes the non-determinism of test executions, which is especially prevalent in highly dynamic frontends. The reason for this is that when writing tests one typically has to wait for certain conditions, such as changes in the DOM to be completed, in order to continue executing the test logic. However, it can be difficult to determine when these conditions are met, i.e., when a quiescent state is reached, leading to the use of timeouts as a workaround. If a condition is not met within a predefined amount of time, the test is considered to have failed. Setting the correct timeout value is a balancing act. If timeout limits are set too low, they may not account for fluctuations in server response times or client computing power, which may lead to an unresponsive user interface. On the other hand, if timeout limits are set too high, tests may take longer than necessary to terminate, preventing fast feedback loops.

Unstable selectors Another challenge in end-to-end testing web applications through the browser is finding *stable* selectors that are resilient to structural changes in HTML documents. During software evolution, a site's markup may change and elements may be moved to different locations in the DOM while still exhibiting the same behavior as before. If there exist tests that use selectors that involve the position of the element in the DOM or that are too general, those tests will likely fail at some point and need to be adjusted. Several approaches address this issue either by using heuristics based on a set of combinatorial rules that define how selectors for specific element are composed [59, 72] or by using constraint solving techniques [14]. These approaches are suitable when mining data from third party websites or in scenarios where testers cannot influence the HTML code directly. This may be the case when testing and development teams are separated, or when using the *Record & Replay* approach, provided for example by tools such as Selenium IDE [93] and WaRR [7], although it has been shown that even recorded tests also tend to break easily [42].

The *design for testing* approach to web application development aims to avoid the above problems by designing and implementing systems in a way that facilitates testing right from the start. A common practice that falls under this paradigm is the instrumentation of HTML elements with `data-*` attributes, as the Cypress testing framework [25] describes in its best practices for writing *stable selectors*. These isolate element nodes in the DOM, allowing the implementation of tests that are likely to be resilient to system changes. Thus, elements can be tagged for testing, allowing testers to create selectors that are not tied to visual properties, the element's position in the DOM, its behavior, or its textual content. In this work, the idea of instrumenting HTML code with special `data-*` attributes is taken up in Chapter 4 in order to simplify the creation of system inputs used by learning algorithms and to reduce flakiness in long-running learning processes.

2.3 Automata Learning

Since test creation can be a time-consuming process, one way to reduce the effort is through the use of model-based testing. This involves generating tests from formal system specifications, such as Unified Modeling Language (UML) diagrams [62]. However, this requires the existence of these specifications. Automata learning, also known as model-based testing without models, aims to solve this problem by inferring the formal specifications of a System Under Learning (SUL) in the form of automaton models. There are two variations of this approach. On the one hand, *passive (automata) learning* refers to the inference of automaton models by analyzing examples of system output, such as system logs, as described in [63]. The drawback of this is that the state space of learned models is limited to what is encoded in the recorded examples, which may only expose partial system behavior. In *active (automata) learning*, on the other hand, a *learner* interacts directly with a SUL via provided interfaces and is therefore able to generate required traces of the system behavior itself. For the rest of this work, the term *"learning"* will refer specifically to the active learning variant.

Active learning was introduced in the context of learning Deterministic Finite Automaton (DFA) from regular languages by Angluin in [8] with the development of the L* algorithm. This algorithm as well as more recent alternatives such as DHC [68] and TTT [48] follow the Minimal Adequate Teacher (MAT) principle where a *teacher* is asked two types of queries. Given a language L and an *input alphabet* $\Sigma = \{\sigma_i, \ldots, \sigma_n\}$, *membership queries* answer the question whether a particular word $w \in \Sigma^*$ is part of L. Based on the teacher's answers to these queries, an automaton model M representing L can be inferred. The second type of query are *equivalence queries*, which answer if $M \equiv L$. If this relation is found to be false, the teacher provides a *counterexample* – a word w where the output functions $L(w)$ and $M(w)$ produce a different output. Counterexamples are then used to refine M by posing additional membership queries in a subsequent learning iteration. The learning process terminates when no more counterexamples can be found through equivalence queries.

2.3.1 Learning Web Applications

Over the years, active learning algorithms have been developed to support a wider range of automaton types, such as Mealy machines [4, 95], System of Procedural Automata (SPA) [36], and even register automata [2]. Frameworks like LearnLib [69] provide implementations of active learning algorithms and abstractions for querying real-world systems in order to learn them. A key concept in this process is the *mapper* [49], which translates the abstract symbols used by learning algorithms into concrete actions for a specific target SUL. Figure 2.2 provides an illustration of the learning process implemented with LearnLib. The *membership oracle* and the *equivalence oracle* provide abstractions for processing membership and equivalence queries, respectively. The membership oracle passes abstract queries to the mapper, which translates them into concrete queries and processes the output from the SUL. Once a model is inferred, it is passed to the equivalence oracle. Due to the black-box nature of active learning, equivalence queries must be approximated through testing in practice. The oracle therefore generates queries using a specific strategy (e.g. random or conformance testing methods such as the (partial) W-Method [23, 34]) and returns a counterexample if divergent output behavior is observed between the model and the SUL. Finally, the counterexample is used to refine the model.

This dissertation focuses on learning web applications, which can be considered reactive systems because they typically do not terminate on their own and respond to interactions,

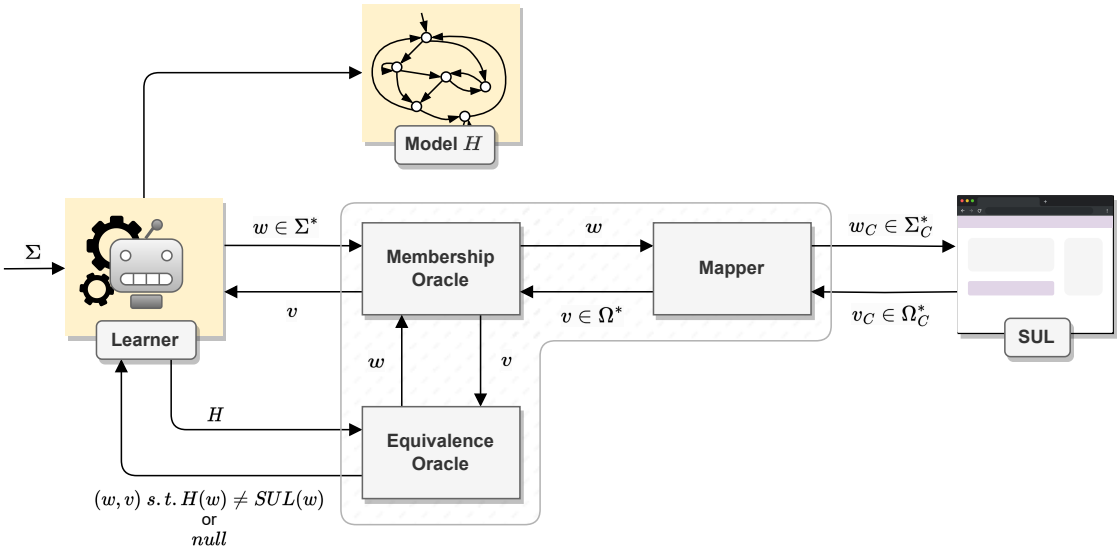


Figure 2.2: Active learning of black-box systems with LearnLib.

such as by updating the user interface. In the past, Mealy machines have proven to be an adequate representation for verifying user-level specific system properties in these types of systems [12, 67, 80, 88, 89]. Browser automation libraries such as Selenium [92] are key to implementing the necessary mappings for web applications. For example, given the string representation of the symbol *"login as admin"*, a script could be written to open a web browser, navigate to the login page, enter the admin's credentials, submit the form, and assert that the login has been performed successfully.

In terms of controlling the quality of a web application during its evolution, Neubauer et al. have shown the importance of *design for verifiability* [77]: By choosing an abstraction for the input alphabet that relates to domain-specific user-level features, the verification of functional properties through model checking is simplified. For example, if the input alphabet for a web application includes symbols for user-level actions like *"create account"* and *"login"* it becomes easier to verify that the application is behaving correctly when these actions are performed. In this context, Neubauer et al. conclude in [80] that a stable alphabet abstraction, i.e., an abstraction where the alphabet has to be adapted only when features are added or deleted, is essential to controlling the system evolution. The reason for this is that a stable alphabet makes it easier to verify that a system is changing in a desired way and ensures comparability between learned models.

2.3.2 State-Local Alphabets

State-local alphabets have been first described by Isberner et al. in [46] and evaluated in the context of learning mobile applications through their user interface as Mealy machines in [38]. The motivation for state-local alphabets arises from the specific trait of systems such as web and mobile applications, that not all inputs are possible to execute at all times. Using this knowledge, the total amount of posed membership queries can be greatly reduced as queries which include sequences that are known to not be fully executable are not posed in the first place. The global input alphabet Σ can therefore be expressed as the union of all state-local alphabets, i.e., $\Sigma = \bigcup_{i=1}^{n} \Sigma_i$, where Σ_i represents the local alphabet of state s_i.

In practice, obtaining state-local alphabets requires an interface from the SUL that

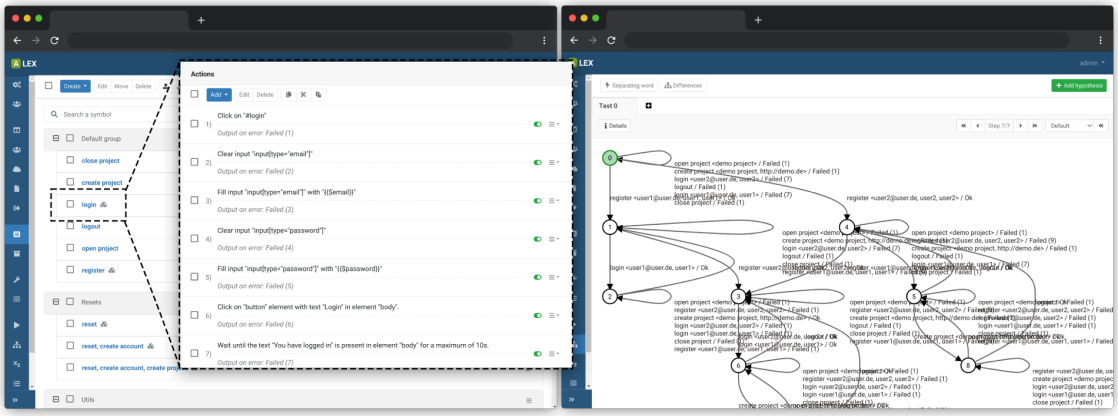


Figure 2.3: Web-based user interface of ALEX: input symbol management (left) and model visualization (right).

can be accessed at any time to retrieve a set of currently available inputs. If at some point during the execution of a membership query, an input symbol σ is not within the set of enabled inputs, the execution is halted and the suffix of the query is answered with the output symbol ω_{undef} which reflects undefined system behavior. As a result, the final model will include a sink state where from each state and for each symbol that is not within the local alphabet, a transition is drawn with the corresponding input and ω_{undef} as the output.

2.4 ALEX

Automata Learning EXperience (ALEX) [6, 11] is a web-based tool that allows users to test web applications via learning-based testing in a simplicity-oriented, low-code fashion. The tool therefore provides a framework for defining and running active learning experiments with different algorithms for learning Mealy machines, and allows users to analyze and compare the results of these experiments in a variety of ways. At first, users define input symbols on an abstract level and implement their mapping as a sequence of browser specific user actions, see Figure 2.3. For most of these actions, users can specify XPath or CSS selectors to indicate which elements to interact with in the web browser via the DOM. With these implemented symbols, traditional end-to-end tests can be modeled as a sequence of input symbols, or learning setups can be configured. In a learning setup, users select a set of symbols as the input alphabet, choose from a set of learning algorithms offered by the LearnLib library, select an appropriate equivalence testing strategy, and select a target web browser. The user is then guided through the learning process in the web interface of ALEX. This includes live updates of the model when new states are detected, an exploratory approach to counterexample search through interaction with the visualized model, and the verification of system properties formulated in linear temporal logic [86] using an integrated model checker. In this work, ALEX is extended to be deeply integrated into modern development workflows by providing collaboration support and APIs to automate the execution and analysis of testing and learning processes from third-party tooling. These extensions are the topic of Chapter 3.

2.5 Language-Driven Engineering

LDE is a software development approach that focuses on the use of DSLs to facilitate communication and collaboration among all stakeholders involved in the development process [94]. This includes domain experts as well as system architects and programmers equally. As DSLs are specialized languages designed to represent and solve problems in a specific domain, they are typically more concise and expressive than general-purpose programming languages, making them easier to use for domain experts. Core ideas of LDE are *horizontal composition* and *vertical refinement*.

Horizontal composition allows separation of concerns by creating distinct DSLs for different purposes in a domain, enabling different experts to work on specific areas. For example, in the context of web applications, horizontal composition could consist of languages for modeling business processes, domain data, and the user interface, as seen in DIME [19], a low-code tool for the model-driven development of full stack web applications.

Vertical refinement involves the creation of a hierarchy of DSLs, each one more specialized and therefore more powerful and user-friendly for the corresponding stakeholder. To take advantage of this, development environments must provide compatibility and interoperability between DSLs.

This design principle allows for improved quality control and change management with respect to the *Archimedean points* in system development [96]: The more purpose-specific DSLs are designed, the better they can be controlled during development and evolution.

In LDE, the focus is on graphical DSLs as they bridge the semantic gap [76] between domain experts and developers. The semantic gap in software engineering refers to the difficulty in accurately and effectively communicating the meaning and intent of software requirements and designs between domain experts and developers. By allowing all stakeholders to participate in the development process, LDE aims to narrow the semantic gap and to improve the alignment between requirements and resulting software. As a result, the risks and costs associated with software development and its evolution are reduced and development efficiency and product quality are improved.

Moreover, LDE revolves around the use of Integrated Modeling Environments (IMEs) which are Integrated Development Environments (IDEs) that support graphical DSLs in the development process. IMEs are optimized for horizontal composition and vertical refinement and provide features known from IDEs for general-purpose languages such as the enforcement of language constraints via syntax checking. Further, they include support for full code generation from (graphical) models, which automates the process of writing boilerplate code and allows stakeholders to focus on more important tasks, such as implementing business logic.

2.6 CINCO

The *CINCO SCCE Meta Tooling Framework*, also known as CINCO [73, 74], is a language workbench [32] that enables LDE by providing the means to develop low-code IMEs. The tool is designed for language designers who want to create domain-specific editors by defining the syntax of graph-based languages and the behavior of generated IMEs at the meta-level (see Figure 1.1). Central to this are three textual DSLs:

- The *Meta Graph Language (MGL)* is used to describe different types of nodes, containers, and edges, as well as their relationships: Nodes and containers can be

connected by edges, and containers can contain nodes and other containers. The MGL also allows for the specification of constraints on the typed graph system, such as the types of nodes that a container type can contain and the number and types of edges that can connect nodes and containers. Moreover, each MGL can be linked to a code generator that generates code from models created in the corresponding IME.

- The *Meta Style Language (MSL)* is used to describe the appearance of elements defined in the MGL. This includes properties such as the shape and color of nodes and containers and the line style of edges.
- The *CINCO Product Definition (CPD)* language is the entry point for any CINCO-based project. It specifies a set of MGL files for which the generated IMEs provides modeling support, as well as meta-information about the product, such as its name and its version number.

Based on these languages, CINCO generates an Eclipse-based IME application that provides the tools necessary to create graphical models that adhere to the language constraints defined in the MGLs.

In an attempt to lift CINCO products to the web, Zweihoff et al. introduced Pyro [112] as an alternative generation target for CINCO products. Instead of an Eclipse-based IME, Pyro-based IMEs can be deployed to the web and offer a multi-tenant capable web application with a focus on real-time collaboration. As a further development of this, CINCO Cloud [10, 111] is the next step in the evolution of LDE, lifting the language workbench as well as generated IMEs to the web as a single application.

Chapter 3

Lifelong Learning of Web Applications

The role and the impact of active automata learning for the Quality Assurance (QA) of web applications has already been demonstrated in previous research [12, 67, 80, 88, 89]. Code-centric frameworks such as LearnLib [69], Tomte [1], and libalf [17], are targeted at programming experts, allowing them to tailor the learning setup to their specific needs and adapt their setups to a wide range of target applications. However, this flexibility is paired with a steep learning curve, tying QA engineers to the framework's language and requiring them to create and maintain learning setups, input alphabets and appropriate mappers in code. With the development of LearnLib Studio [15], a first step in the direction of making automata learning practically available for a non-expert crowd has been made using a graphical Domain-Specific Language (DSL) to model learning in a dedicated Integrated Modeling Environment (IME). Implemented using the jABC [79], the predecessor of CINCO [74], developers can model learning setups in a process-driven fashion using a graphical modeling environment. Drawbacks of this approach include the lack of collaboration support, the inability to integrate with modern development stacks and the requirement to develop system-specific mappers manually.

This chapter presents the essence of the attached publications [9, 12, 13, 98] with a focus on domain-specificity and its benefits for the framework built around ALEX [6, 11]. ALEX is a web-based, low-code tool for active automata learning of web applications, tailored to address the pain points of related tools and focused on ease of use and integrability. The tool is the backbone of the simplicity-oriented *lifelong learning* framework that enables fully automated continuous quality control of web applications via learning-based testing. It compromises on flexibility in favor of simplicity [64] that the previously mentioned tools lack, and in doing so it leverages

Domain-Specificity By narrowing down the domain, tools can be tailored to the specific needs and challenges of that domain, and the focus can be put on making relevant tasks as simple as possible for users to perform. For example, ALEX eliminates the need for users to create the infrastructure to run automated tests in a web browser and provides an all-in-one solution for common tasks related to learning-based testing of web applications without the need for third-party tools. All aspects such as alphabet definition, learning process configuration and model verification are seamlessly integrated and designed to work together and to support each other in a low-code manner, only requiring users to have knowledge of their domain: Hypertext Markup Language (HTML) and Cascading Style Sheets (CSS).

Integrability An integral part of modern software development is the automation of repetitive processes using Continuous Integration and Continuous Deployment (CI/CD). Leveraging CI/CD, learning-based testing can be fully automated to ensure software quality during development. As a central management and execution engine, ALEX is designed to integrate well with CI/CD vendors by offering public Application Programming Interfaces (APIs) that can be used, for example, to trigger learning

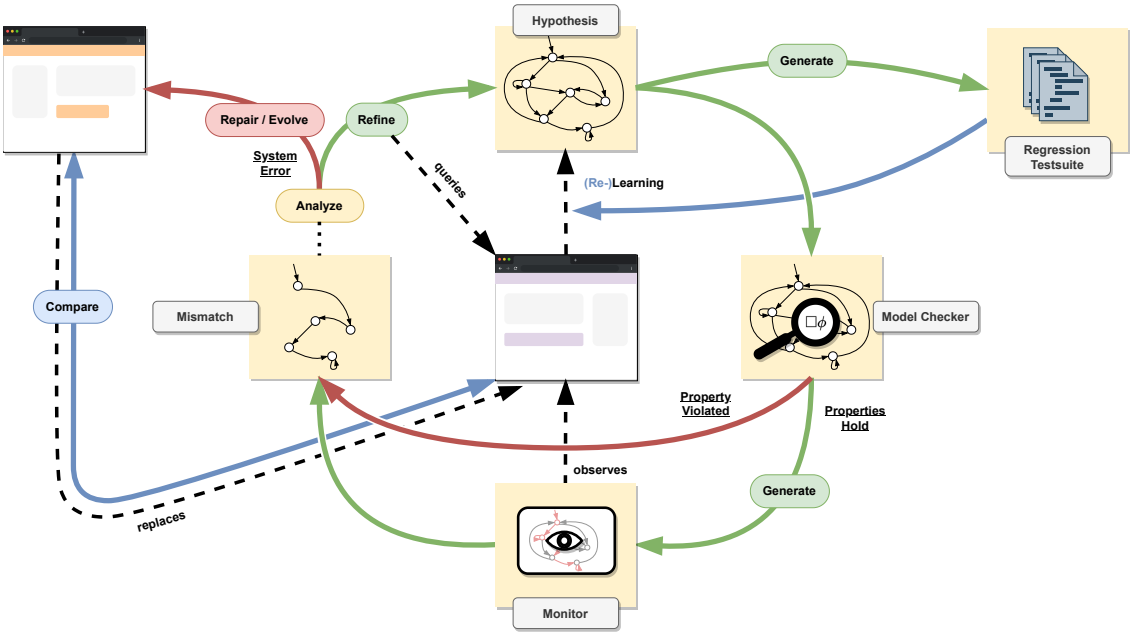


Figure 3.1: Illustration of the lifelong learning cycle, see [13].

processes from within CI/CD pipelines. Additionally, an event-driven notification system allows external services to listen for application-specific events and respond to those events in a desired manner. In combination, these interfaces allow developers to control ALEX programmatically and to extend its functionality in a service-oriented way.

Collaboration Unlike local-first software solutions, web applications can provide a better collaboration experience through real-time updates, accessibility regardless of physical location, and centralized data management (single source of truth). Thus, ALEX as a web application itself encapsulates all data relevant to learning target systems in *projects* that can be shared with team members, allowing different people with different responsibilities to work on different quality control aspects. For example, this makes it possible to distribute the definition of input symbols and tests, their concrete implementations and the formulation of Linear Temporal Logic (LTL) [86] properties among multiple individuals.

Lifelong learning has already been introduced as an abstract concept in [16] and has been implemented, extended and demonstrated on an adaptive cruise control system in the automotive context in [13]. The process can be illustrated as a cycle, shown in Figure 3.1. In this cycle,

- active automata learning is used to learn formal models, i.e., Mealy machines from web applications that represent their user-level behavior,
- system properties are formulated in temporal logic and verified automatically on inferred models using model checking,
- runtime monitors use learned models as "sensors" to observe the behavior of a web application to detect potential behavioral defects over longer periods of time,
- mismatches found by model checkers or runtime monitors lead to automated model

refinements in case the model is not accurate enough or to system refinements in case actual system errors are revealed,

- regression test suites are generated using model-based testing, enabling faster re-learning processes in subsequent software iterations, and
- the evolution of the system is controlled by comparing models of two subsequent iterations, thereby verifying that the underlying system changes in a desired way.

The framework only requires users to define alphabets, model their mappings, formulate system properties in LTL, and configure learning setups through a web-based interface. Everything else can be fully automated. Human intervention is only required when mismatches occur that are not suitable for model refinements and for adapting the input alphabet and its mapping to system changes. The following sections provide a brief overview of how ALEX aims to make the phases of the lifelong learning cycle accessible for a non-expert crowd and how it eases the integration with third party tooling for automation purposes. Please note that user-level monitoring of web applications is an open research topic. It is not supported by ALEX and will be addressed in future work, see Section 6.3.

3.1 Alphabet Modeling

Learning real systems typically requires users to define alphabets at an abstract level that is used by learning algorithms, and at a concrete level so that the target system can be interacted with. In ALEX, the concrete level is implemented as a sequence of user-level *actions* performed on the user interface through the web browser, as depicted in Figure 2.3. There, the abstract input symbol *login* is mapped to a sequence of actions that fills two input fields with predefined credentials, triggers the submission of the corresponding form, and then verifies that the string *"You have logged in"* is present on the website. To model such sequences, users can choose from a wide range of configurable actions that represent common user-level interactions, such as clicking elements, filling in input fields, and submitting forms. Most of these actions require only the specification of a CSS or an XPath locator pointing to elements in the Document Object Model (DOM) of the website. This benefits users by eliminating the need to familiarize with any specific browser automation tools and even requires no knowledge of other programming languages. During a learning process, the mapper interprets the configurations for all input symbols of a membership query and executes the corresponding code on the target web application.

Regarding the output alphabet, a default but configurable abstraction is provided where the response of a web application, which is tightly coupled to the modeling of the system inputs, is interpreted and mapped to a string for use by the learner. If the sequence to which an input symbol is mapped can be executed successfully, which means that a user could perform this sequence without any issues as well, the output of the system is interpreted as `OK(<MESSAGE>)`, where `<MESSAGE>` is a possibly empty, user-defined string that allows to add more semantics to the learned models. On the other hand, if the execution fails at any point, the system output is interpreted as `FAILED(<MESSAGE>)`. Again, a custom message can be specified for the action that failed to provide a more semantic output for the automaton model.

3.2 Model Learning

To keep the hurdle to learning models as low as possible, ALEX relieves users of the task of instantiating a custom learning setup. Instead, users can configure a predefined setup

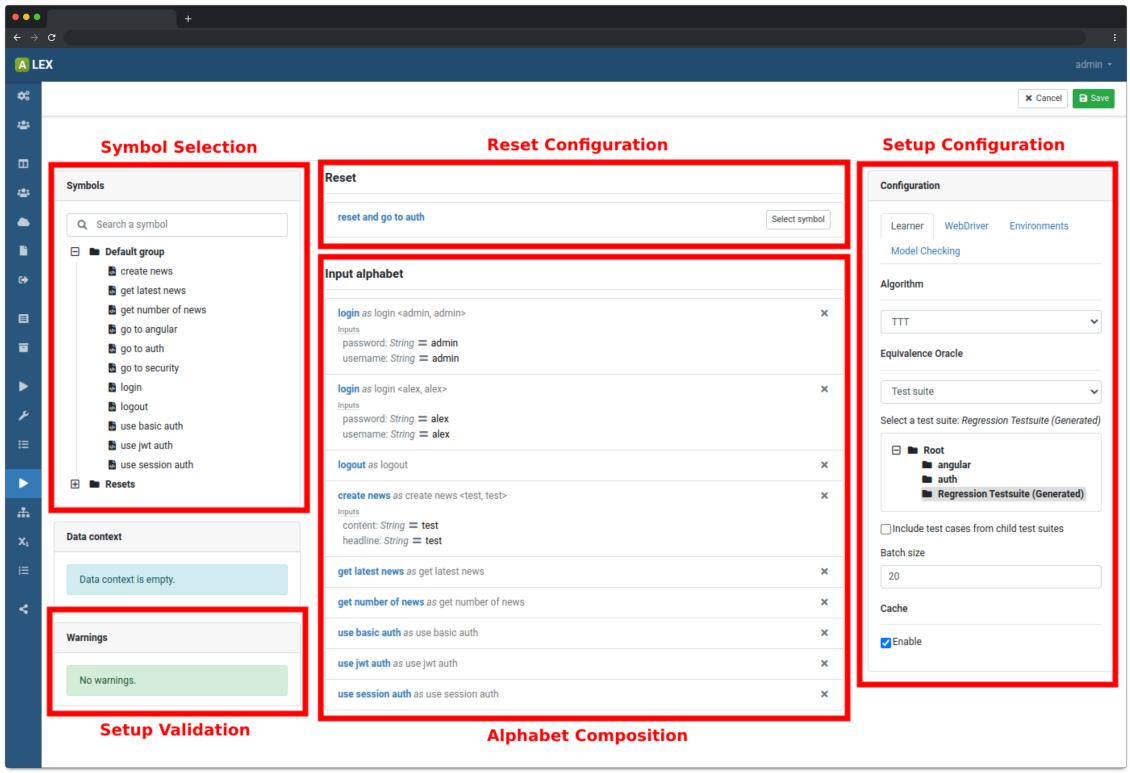


Figure 3.2: Configuration of a learning setup in ALEX.

with sensible defaults in a single view, as shown in Figure 3.2. Users only need to assemble the input alphabet from a set of previously modeled input symbols and configure aspects such as the system reset, the learning algorithm, the equivalence testing strategy, a web browser to access the target application, and optionally, a set of LTL properties that are automatically verified on the learned models. A validation view provides additional visual feedback on the completeness and correctness of the configuration.

Once started, the progress of a learning process is displayed in real time: In addition to intermediate models, various statistics such as the number of membership queries and executed symbols and the duration of the process are collected and displayed. Because statistics and learned models are persisted automatically, the tool simplifies the design and execution of case studies and benchmarks, such as performed in [12].

Moreover, users can manually search for counterexamples in an exploratory fashion by interacting with learned models within the tool. This has proven to be particularly useful feature in teaching automata learning theory, allowing users to act as teachers in the Minimal Adequate Teacher (MAT) principle, see Section 2.3. For this purpose, learned models are visualized and users can assemble words to test against the target system by clicking on the edge labels of the model. After the output of a potential counterexample has been validated against the actual system output, visual feedback is provided indicating whether the assembled word is suitable for model refinement.

3.3 Model Verification

Inferred automaton models by themselves are a mere aggregation of observations and have no inherited understanding of correct or incorrect system behavior. For this reason, ALEX integrates the LTSmin toolset [50], which includes a model checker for LTL formulas

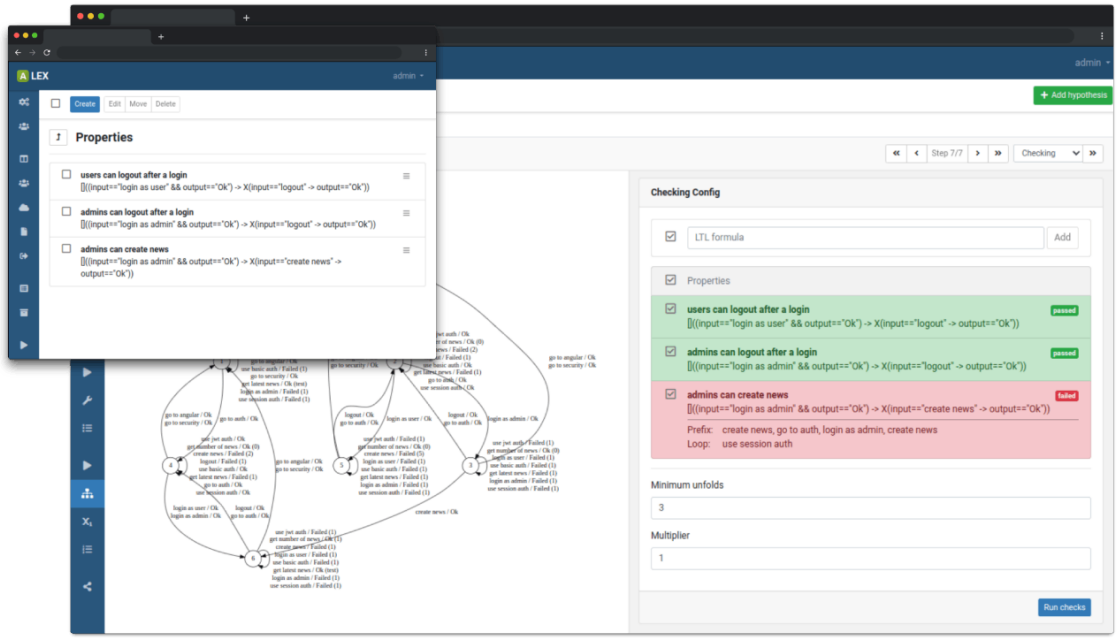


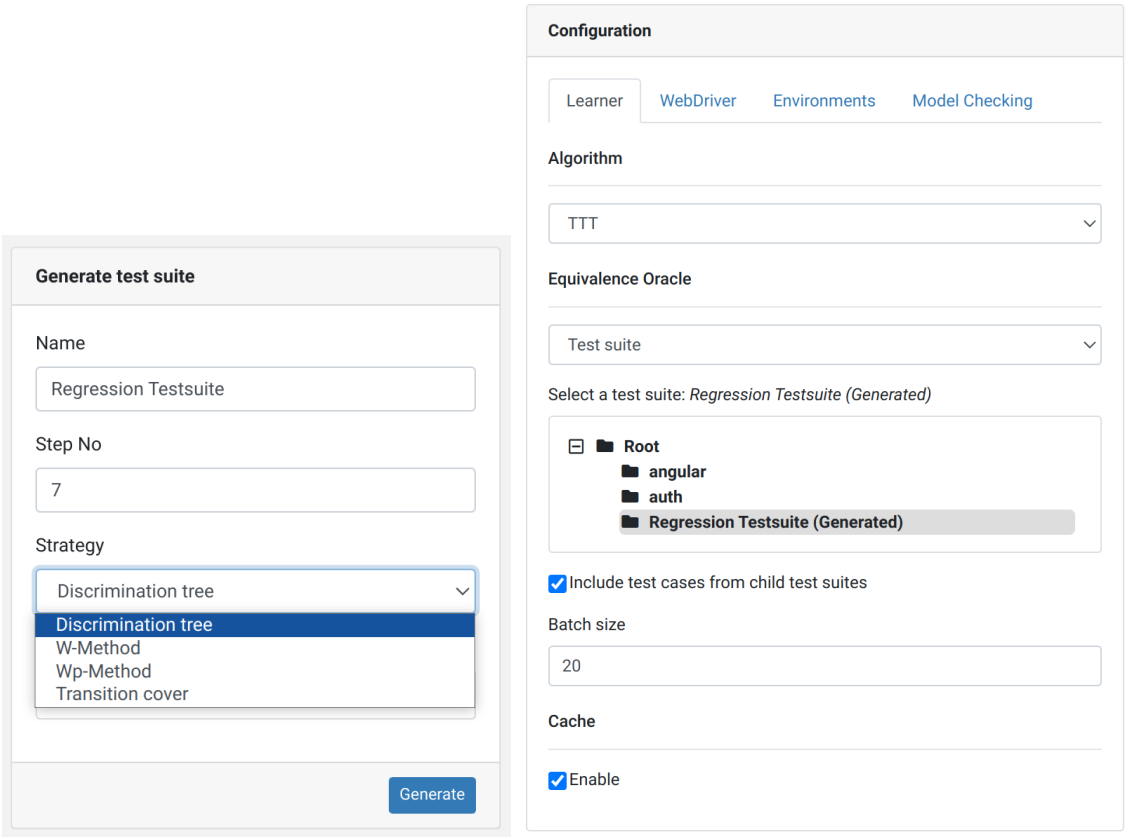
Figure 3.3: Verification of LTL properties in ALEX.

and a corresponding textual DSL in which these formulas can be described. On top of this, AutomataLib [39] provides the necessary parser for LTL formulas and a wrapper that allows it to verify Mealy machines using formulas that include input and output labels in propositions separately. For example, the following formula translates to *"Users should be able to logout after a login"*:

```
[]((input=="login as user" && output=="Ok") ->
    X(input=="logout" -> output=="Ok"))
```

In the tool, these formulas can be created, managed, and later verified on learned models in a dedicated view illustrated in Figure 3.3. Note that to improve usability, only syntactically correct formulas can be persisted and used for validation, and users will be notified if syntactic correctness is violated. The upper left part shows the management overview, while the other part shows the result of a manually triggered model checking execution on a learned model. When a system property is verified, the corresponding formula is highlighted in green, otherwise it is highlighted in red, and a counterexample is provided at the same time.

Users benefit from the model checking capabilities of ALEX in two ways. First, as indicated in Section 3.2, predefined LTL formulas can be specified during the configuration of a learning process. These are automatically verified on the final model and the results are presented visually. Second, user-created formulas can be used to perform black-box checking [84], an approach that uses model checking to perform equivalence tests on intermediate hypotheses. If some properties are violated, the model checker provides counterexamples that are first tested against the actual system under learning and then used for model refinement if the output of the model and the output of the target system differ. This approach is particularly useful in the first few iterations of the learning loop, as it saves expensive membership queries before exhaustive counterexample searches are performed. As a result, the overall runtime of the learning process can be reduced, providing faster feedback to developers and QA engineers.



(a) The widget for generating tests.

(b) The configuration form for selecting a test suite as an equivalence oracle.

Figure 3.4: Generation of regression test suites in ALEX.

3.4 Test Generation

Model-based testing is a research area that aims to generate system tests from formal models, e.g., with respect to a certain coverage criterion. In the lifelong learning cycle, model-based testing is leveraged in two ways:

- Generated tests can optimize equivalence tests in subsequent development iterations. As long as the behavior of the target system under does not change inherently, generated tests encode information that can be used for a more guided counterexample search. This way, ideally, a few core tests can help a learner build a hypothesis more quickly, thereby reducing the overall learning time.
- By running generated and manually created end-to-end tests before performing time-intensive learning processes, a fail-fast mechanism can be implemented to provide fast feedback in case of regressions.

Key to the effectiveness of this is that the generated test suites must be relatively compact compared to other strategies, such as the W-Method or the Wp-Method, while maintaining enough information about the system behavior. As presented in [13], ALEX uses discrimination trees provided by the TTT algorithm [48] to generate these test suites in an efficient manner. Discrimination trees have already been introduced in the context of automata learning by Kearns and Vazirani [52] to store observations more efficiently than in observation tables used by Angluin [8]. They have shown to represent the behavioral

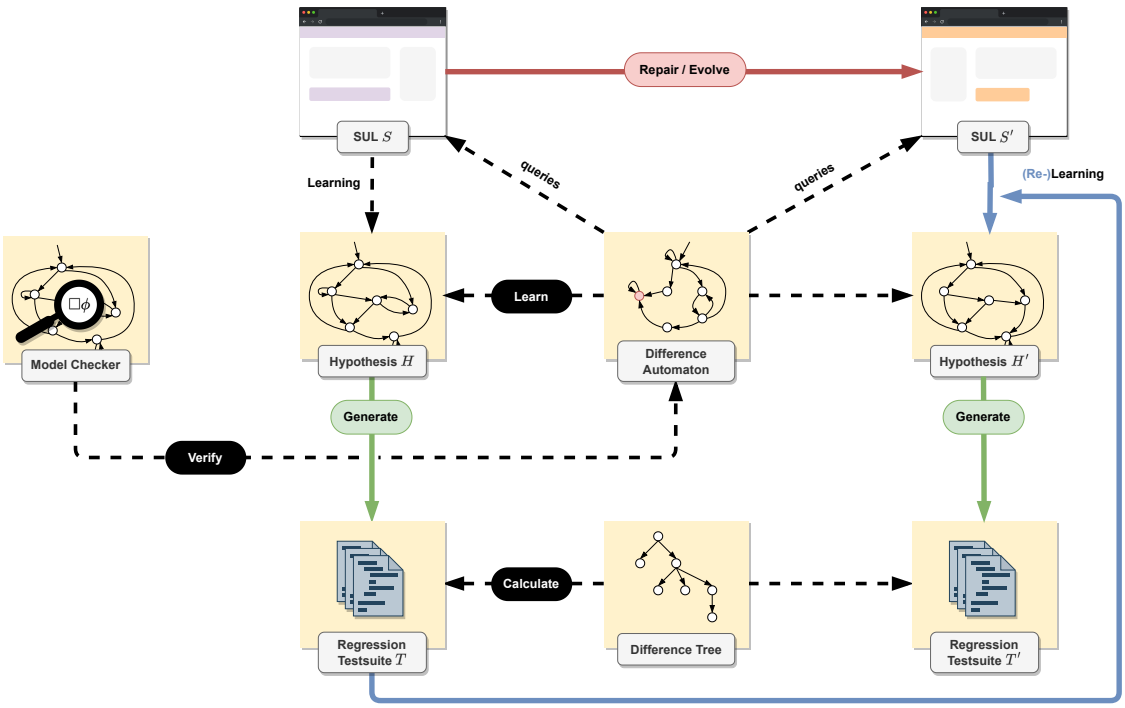


Figure 3.5: Evolution control in lifelong learning, see [13].

essence of the system and allow to reconstruct the model with only a few additional membership queries. An example of a discrimination tree for a seven-state automaton is displayed in Figure 3.7. Test suites that guarantee at least state coverage are built as follows: For each leaf representing a system state, a set of tests can be generated by using the access sequence of the state as a prefix and each discriminator from the leaf to the root independently as a suffix. In the case of the discrimination tree shown in Figure 3.7, this accumulates to 21 tests. This compares to 336 tests using the W-Method and 190 tests using the Wp-Method, clearly demonstrating the advantages of this approach. To generate a test suite from an existing model, users are presented with a form that allows them to choose between different test generation strategies, see Figure 3.4. In addition, the same test suite can be selected to be used by the equivalence oracle for re-learning purposes when configuring a learning setup, see Figure 3.4b.

Generated test suites are optimized for model reconstruction. However, they tend to appear arbitrarily assembled to end users, as they may not cover intuitive user-level paths that may be useful for verifying specific acceptance criteria. Because in ALEX allows tests to be modeled as a sequence of input symbols, where for each input the expected output can be specified, users can enrich test suites with manually created tests that make more sense to them. This also provides a better on-boarding experience for people unfamiliar with automata learning. Initially, users model input symbols and compose tests, and eventually use the same symbols to learn their target application and use the created tests to improve the learning process.

3.5 Evolution Control

A challenge in software development is to ensure that systems evolve in a controlled manner. Typically, regression testing is performed to ensure that changes to the software do not

break existing functionality. However, this typically focuses only on specific areas of the software that are likely to be affected by the changes. This can create a false sense of security, as there is no guarantee that the changes will not have side effects on other areas or on the overall user-level behavior of the application. Learning-based evolution control provides a holistic view of changed user-level behavior by comparing learned models from two successive software releases. This can be used not only to detect that something has changed, but also to generate traces of divergent behavior, and even to verify that the system has changed in the intended way using model checking. The approach is illustrated in Figure 3.5. Means of comparing two systems introduced in [13] are *difference automata*, which represent differences as a graph and *difference trees*, which are tree-based representations of differences computed from generated regression test suites. Both are discussed in the following sections, where a difference between two Mealy automata is defined as an input sequence that, when executed by both models, produces a different output sequence. A prerequisite for this approach is a stable alphabet abstraction over multiple software iterations, which is manifested in the fact that the models in question must use the same input alphabet. Otherwise, if there is an input symbol σ that is included in one alphabet but not in the other, the shortest trace of divergent behavior would be σ itself.

3.5.1 Difference Automata

Difference automata, in the context of this thesis, are Mealy machines that capture the difference in the input-output behavior of two systems. They can be built using standard learning algorithms where membership queries are posed to both target systems or their respective models simultaneously. Given two Mealy machines M_1 and M_2 and their respective output alphabets Ω_1 and Ω_2, queries are processed step by step and the outputs are analyzed. If, for a given symbol σ_i, the outputs of $M_1(\ldots \circ \sigma_i) = \ldots \circ \omega_{i_{M_1}}$ and $M_2(\ldots \circ \sigma_i) = \ldots \circ \omega_{i_{M_2}}$ differ, the output at position i will be answered with $\omega_{i_{M_2}} \leftrightarrow \omega_{i_{M_2}}$. Moreover, the remaining inputs of the query are not processed further and are instead answered with a special output symbol $undefined \notin \Omega_1 \cup \Omega_2$. The idea is that if the models produce a different output for the same input, what comes after the first occurrence of divergent output is not relevant from the QA engineer's point of view. Due to the filter, difference automata will have a sink state if M_1 and M_2 differ, where all transitions showing divergent input output behavior point to. All paths from the initial state that end in this sink state are traces of divergent behavior. In the other case, if no occurrence of divergent input-output behavior is observed, the models of the two systems are behaviorally equivalent and the difference automaton converges to M_1 and M_2. For usability reasons, ALEX displays an empty automaton in the latter case, since it better represents the absence of behavioral differences.

An example of a difference automaton can be seen in Figure 3.6, which resulted from learning two versions of a sample web application for publishing news articles used to teach students about authentication and authorization mechanisms. State 4, which is highlighted in red, denotes the sink state, where all reflexive transitions have been removed for clarity. As it can be seen, differences in user-level behavior have been observed, represented by the highlighted transitions $(3, 4)$ and $(6, 4)$ leading to the sink state 4. In the first case, administrators cannot log out of the application after creating a news article, although this was possible in a previous version, as indicated by the transition label "$Ok \leftrightarrow Failed(1)$". In the second case, users can create news articles, even though they were not allowed to do so in the previous version, as indicated by the label "$Failed(5) \leftrightarrow Ok$". Analyzing whether these paths represent desired changes remains a task for application experts and can be

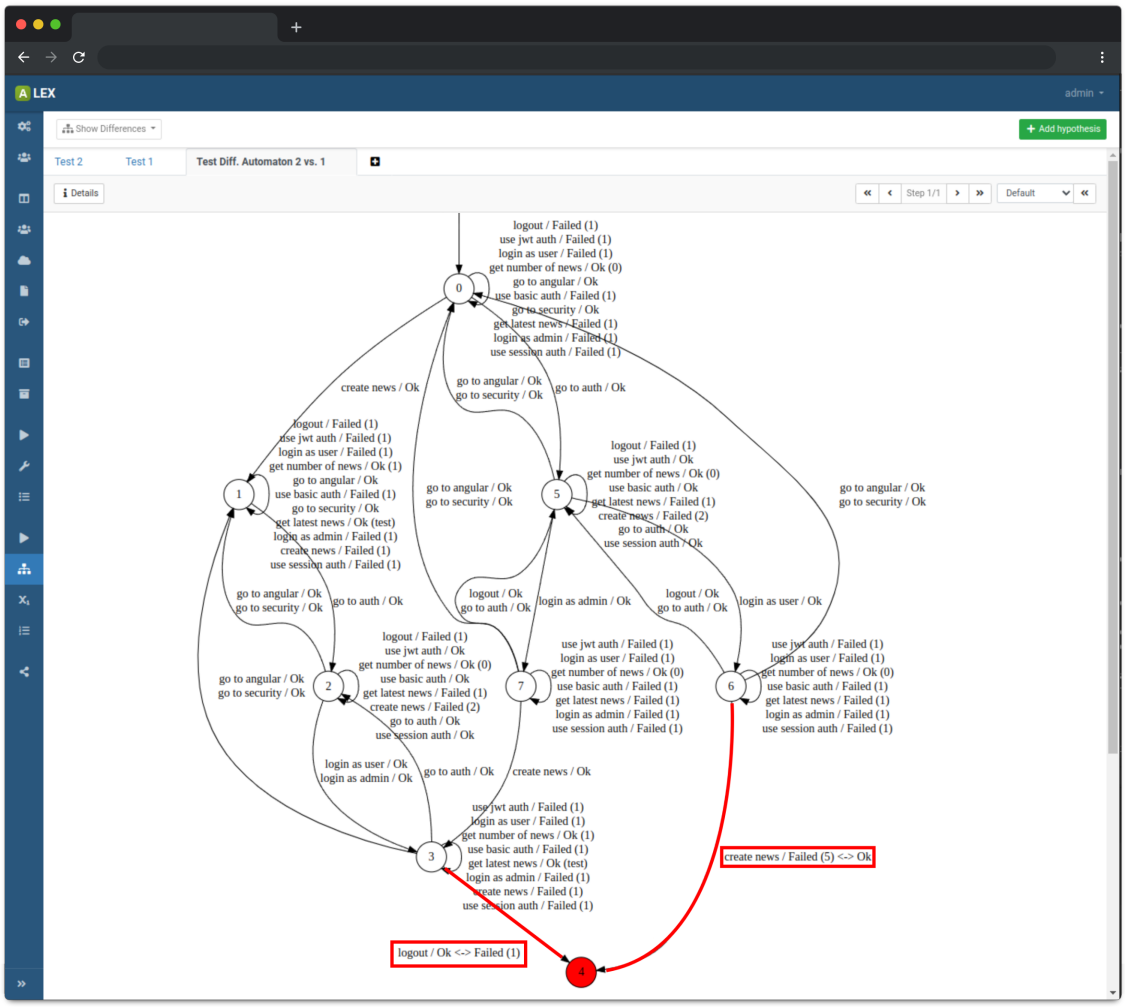


Figure 3.6: A difference automaton for a sample web application in ALEX.

verified manually or using model checking by writing appropriate formulas that respect the output alphabet of the difference automaton.

Like other learning methods, this approach does not claim that inferred difference automata are correct or complete. Due to the black-box nature of end-to-end testing, there is no guarantee that all differences will be found, as there may always be an input sequence that was not posed during the learning process and where the two systems in question produce a different output. Furthermore, interacting directly with the instances of the systems to obtain the difference automaton can be a time-consuming process and therefore unreasonably expensive to execute. If instead models of the systems already exist, e.g., because they have been learned as part of an automated quality control process, obtaining the difference automaton is a matter of performing a learning process against the models instead of the real systems. In this case, conformance testing strategies such as the W-method of the Wp-method can be used as equivalence oracles to ensure that the observed behavior of both systems is respected when creating the difference automaton.

3.5.2 Difference Trees

While difference automata provide an overview of all paths leading to divergent observations, they are not very intuitive for non-experts. Thus, the core idea of *difference trees* is to have

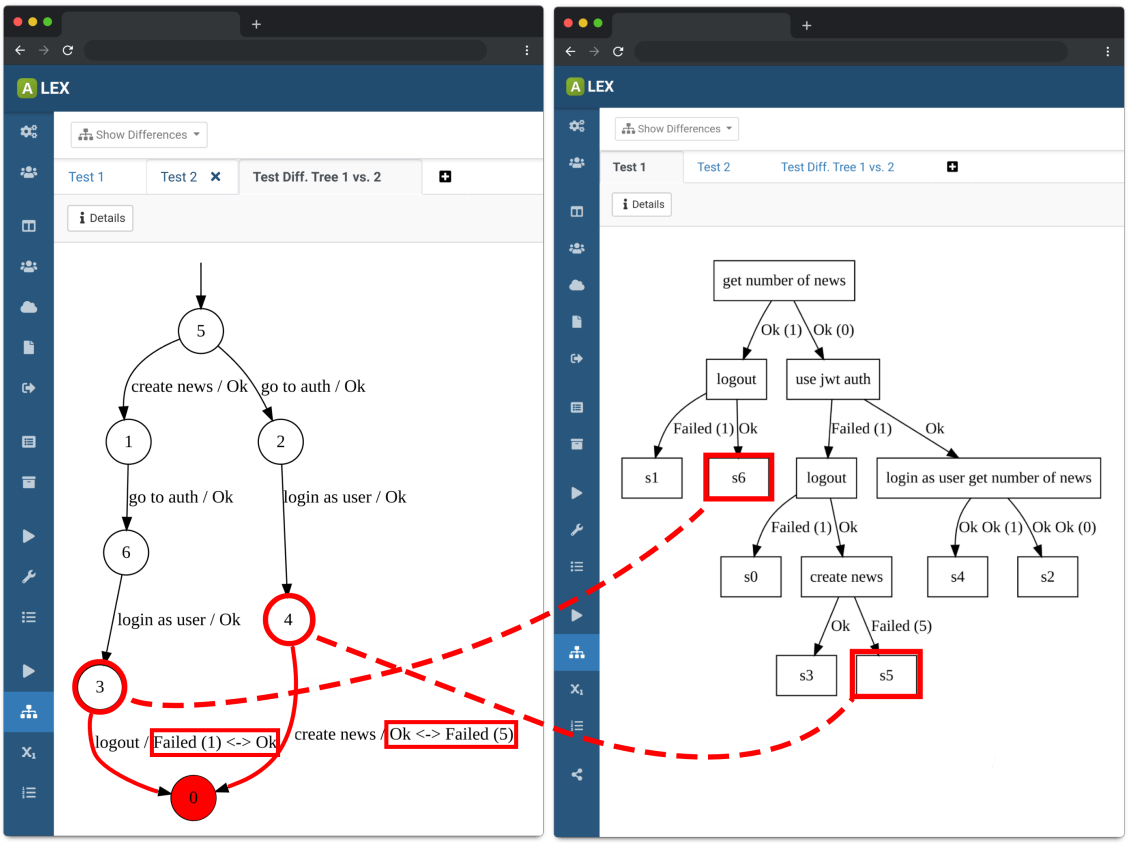


Figure 3.7: A difference tree for a sample web application in ALEX. The difference tree on the left and an underlying discrimination tree on the right.

a compact representation of differences that is easy to understand. In fact, all paths of the difference tree can also be found on the difference automaton [13]. Difference trees are Directed Acyclic Graphs (DAGs) with a root and a leaf node, where each path from the root to the leaf represents a trace of divergent input-output behavior. In this graph, the root represents the initial state of the system, and transitions between nodes are labeled with the input and output of the system using the same notation borrowed from Mealy machines. An example of a difference tree for the application introduced in Section 3.5.1 is illustrated in Figure 3.7 on the left. From the initial state, two paths lead to the leaf labeled with 0, where divergent behavior is visible at the transitions $(3, 0)$ and $(4, 0)$. In the example of $(3, 0)$, the label *"logout / Failed (1)* $\leftrightarrow$ *Ok"* indicates that in one version of the system, a user cannot perform a logout after the previous steps, while in the other system, the logout succeeded. Similar to the difference automata in Section 3.5.1, the DAG has to be analyzed by manual or automated means to verify that the system has evolved in the desired way.

Difference trees are constructed from two test suites generated from two given models M_1 and M_2, which have to share a common input alphabet Σ. First, the DAG is initialized with a single root node indicating the initial system state. Each test corresponding to an input sequence $w \in \Sigma^*$ from both test suites is posed to the query filter used for learning the difference automata. If w produces a different output sequence for both models, the suffix with $undefined$ outputs is removed, and w is inserted into the tree. During the insertion, two nodes are merged if the input and output labels of their prefixes are the same to obtain the DAG structure. As a result, all paths from the root to the leaf in the

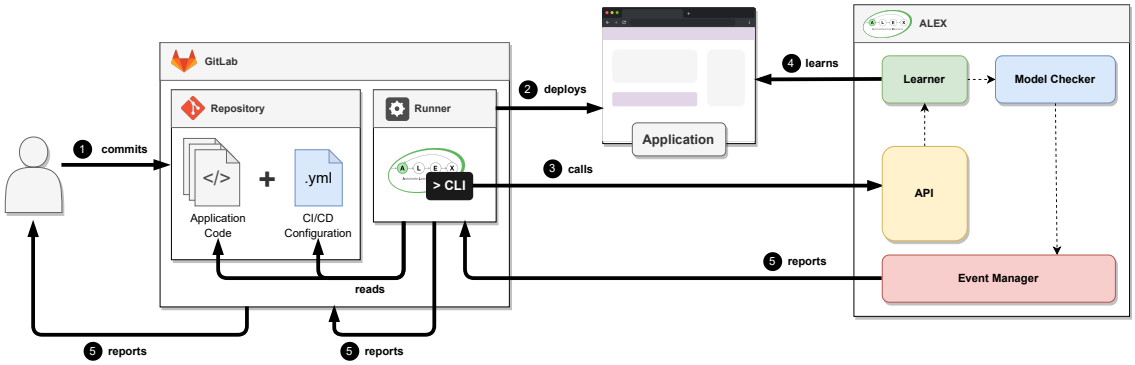


Figure 3.8: System architecture for the automation of the lifelong learning cycle.

DAG are traces of divergent system behavior.

The number of traces found depends mainly on the size of the test suites used to build a difference tree. Smaller test suites are likely to result in DAGs with fewer branches, as fewer paths of the opposite model are covered, possibly resulting in fewer differences being detected even though they may exist. For the difference tree in Figure 3.7, the discrimination tree provided by the TTT algorithm has been used for test generation, resulting in two paths of divergent observations. In contrast, using the Wp-Method for test generation instead results in a DAG with seven paths, and using the W-Method results in nine paths where divergent input-output behavior has been found. Although the latter two strategies seem to be more suitable since more traces are found, in this particular case they show the same error in different forms. This means that multiple paths in the application lead to the same misbehavior, i.e., users cannot perform a logout although they should be able to. It remains to be seen, through future work and further case studies, how well this can be generalized to other applications as well.

3.6 Automation

Automation is an indispensable part of modern software development. It gains particular relevance in the context of DevOps, where a special emphasis is placed on close collaboration between software development (Dev) and system administration (Ops) teams [26]. The means to automation in DevOps are *continuous integration* and *continuous deployment* (CI/CD), which refer to a set of practices that ensure that code is automatically tested with every change and that tested code is continuously deployed. Typically, such processes are described in so-called *pipelines*, which can be either textual, usually using YAML or JSON, or graphical, such as the Rig DSL [98], and which are executed by CI/CD providers such as GitLab. Software testing is an integral part of such automated pipelines, because the earlier errors are detected in the development process, the sooner they can be fixed by the responsible teams.

To exploit the full potential of the lifelong learning cycle, and to make it compatible with DevOps practices, the cycle has been implemented in a CI/CD pipeline authored using Rig, where the ALEX tool acts as a central environment for the execution of testing and learning processes. The tool stores all application-related resources such as symbols, mapper implementations, tests and temporal logic formulas. In addition, the framework relies on GitLab as a project management tool, code repository, and coordinator of the lifelong learning cycle using its CI/CD capabilities. A schematic representation of the automation architecture is shown in Figure 3.8. The process is as follows:

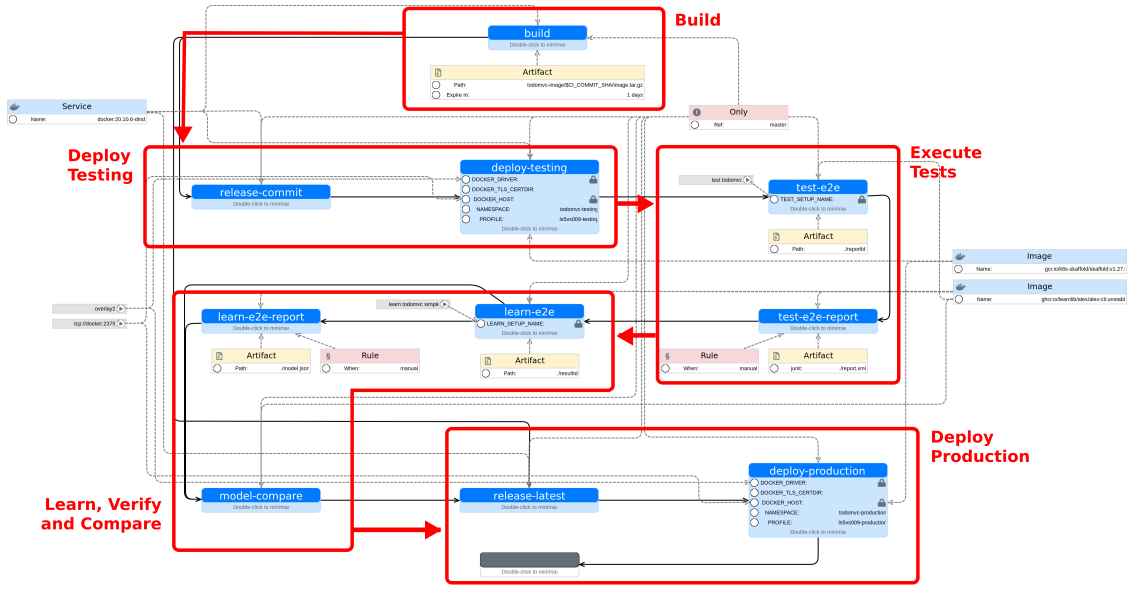


Figure 3.9: A model for a CI/CD pipeline for lifelong learning with ALEX in Rig.

1. Developers commit changes to the application code and the CI/CD configuration file to the repository provided by GitLab.

2. Internally, GitLab recognizes the presence of the CI/CD configuration file and starts a *runner*, i.e., an isolated environment which interprets the configuration file and builds and deploys the application to a test environment.

3. As a part of the process, the runner uses a command line tool that has been developed for ALEX, which provides access to its main features and triggers testing and learning processes through the Representational State Transfer (REST) API of an actively running instance of ALEX.

4. Once initiated, ALEX performs testing, learning and verification processes on the system and its learned model respectively, and finally notifies the internal event manager about the outcome of the process.

5. As a result, feedback on the success of the operation is propagated from the event manager back to the user. So, the event manager calls the runner, which in turn interacts with GitLab which marks the pipeline as successful or failed, which then triggers the notification to the user. If a previous run exists, the models learned in those processes are compared by using the approaches described in Section 3.5.

Figure 3.9 depicts a graphical model of a CI/CD pipeline that implements the lifelong learning cycle for GitLab CI/CD. It is general purpose enough to be adapted to various web applications with only minor modifications to the requested Uniform Resource Locators (URLs) of the REST interface of ALEX. Once a software project includes the CI/CD configuration file generated from the Rig model in its code repository, all QA measures are completely automated.

In total, the pipeline consists of ten sequentially executed steps, starting with building the application, deploying it to a test environment, running testing and learning processes against that environment, and finally deploying the application to a production environment.

In this pipeline, the execution of end-to-end (regression) tests acts as a filter for long-running learning processes: If core tests are already failing, learning processes are skipped to provide fast feedback to stakeholders. On the other hand, if all tests pass, learning and verification processes, as well as evolution control mechanisms described in Figure 3.5, are performed. Note that, due to possible time constraints for pipeline execution, long-running processes are executed in a non-blocking fashion to free up resources used by the CI/CD provider. Since test suites can become quite large over time, and since learning processes can take up to several hours or even days to run, the event manager in ALEX notifies GitLab when these processes are finished so that pipeline execution can continue.

In addition, the pipeline implementation leverages GitLab's project management capabilities. If a problem occurs during the execution of the pipeline, for example if tests cannot be executed, ALEX reports this back to GitLab, whereupon the developers who committed the changes to the repository are notified and issues are created in the project management system. Issues include traces of observations where properties could not be verified, or traces of divergent behavior related to the last iteration in the form of difference automata. This way, all stakeholders are involved in the development process without having to actively use the ALEX tool.

Chapter 4

Learnability-by-Design

Establishing a stable alphabet abstraction, i.e., an abstraction that rarely needs to be adjusted during system evolution, and implementing appropriate mappers to interact with a System Under Learning (SUL) are the main challenges in learning real systems and have to be considered for each application individually. ALEX eliminates this pain by providing a unified, low-code environment for creating setups to learn web applications, which includes the definition of the alphabet, the implementation of mappers, the configuration of learning processes, and the analysis of inferred models. In practice, however, keeping these resources in a separate tool, figuratively speaking, far away from the source code of the target application, introduces certain drawbacks. Typically, application code is managed using version control systems such as Git or SVN, which include features such as branching and merging that cannot easily be reflected in the tool. As development teams and the complexity of a project grow, managing learning artifacts in a separate tool becomes an infeasible practice. Moreover, adjustments to the alphabet and corresponding mapper implementations can usually only be made and tested after changes to the SUL have been deployed, e.g., to a test environment. If changes are not properly communicated, this puts quality assurance teams on spot to determine how the system has evolved and to update learning artifacts accordingly. In development environments where applications are deployed multiple times a day, this can also lead to frustration as developers may be forced to deploy untested code to production due to time constraints. As a result, learning artifacts and the system state can diverge over time, making it difficult to keep up with system evolution and introducing risks to application security.

Learnability-by-design describes a framework that addresses these issues by embedding alphabet definition and mapper implementations directly into the application source code. It therefore provides a standardized solution that can be applied to a wide range of web applications by exploiting their specific traits. Key aspects of the framework include:

- The *instrumented HTML (iHTML)* Domain-Specific Language (DSL), which extends Hypertext Markup Language (HTML) with a set of `data-lbd-*` attributes. These attributes do not change user-level behavior, the appearance of the application in the browser, or the semantics of annotated elements. Instead, they specify input symbol definitions and mapper implementations that enable fully automated learning of deterministic approximations of the application behavior.

- A setup where a learner interacts with a web application without prior knowledge of system inputs by interpreting instrumented HTML code. This approach takes advantage of state-locality [46] in automata learning: In each system state, only a limited set of actions is available given by the Document Object Model (DOM), thereby reducing the number of queries posed to the system. Because system states depend on the DOM with respect to the interaction history, learned models reflect interactions with the user interface of a web application.

- The tool *Malwa*, which facilitates the learning of instrumented web applications

through a minimal, web-based user interface that, in an ideal scenario, only requires users to specify the Uniform Resource Locator (URL) of the target application. If the tool's presets are not sufficient, users can configure aspects such as browser dimensions, the equivalence testing strategy, form and input test data, and the system reset mechanism.

The remainder of this chapter is structured as follows: Section 4.1 discusses the relevance and the implications of using the DOM for automated active learning of web applications, Section 4.2 highlights the benefits of the instrumentation language, Section 4.3 describes the conceptual means to learn instrumented web applications, Section 4.4 gives a technical overview of the iHTML language, Section 4.5 introduces the Malwa tool, and finally, Section 4.6 demonstrates the approach with a practical example.

4.1 Overview

A prerequisite for applying active automata learning is that the systems are *reactive*, i.e., they take some form of input and produce an observable output in response. Web applications are highly reactive systems that users can interact with directly through the web browser, often triggering an immediate response such as loading a new page by clicking a link or changing only parts of the DOM dynamically on the same page. Due to their reactive nature, Mealy machines are commonly used to represent their behavior [12, 67, 80, 88, 89], which by their definition are regular, finite and deterministic automata.

Web applications have the characteristic property that users always have only a limited set of actions available to them via the browser-based user interface. This can be exemplified by the behavioral models depicted in Figure 4.7 and Figure 5.3, where for each state there are only outgoing transitions for the possible actions a user can perform in that state. Typically, web applications consist of different *pages* that display different content and thus offer different functionality. Pages can be understood as actually distinct HTML documents, or in the case of single-page applications, pages can also result from dynamic changes to a single document and can therefore also be referred to as *view* or *state*. The set of possible actions on a page is given by its DOM.

Learning web applications via the web browser requires programmatic interactions with the DOM of a page to simulate user events on visible elements. For this reason, instrumentation is also applied at the HTML level to specify which elements can be interacted with and how they can be interacted with. iHTML as a textual DSL therefore standardizes the way how learning algorithms interact with elements on a page in an automated way. In this context, the key concepts of iHTML are the following:

- *Abstraction* – iHTML reduces the total interaction space on a page to a few instrumented elements of interest within its DOM, see Figure 4.2. This gives developers control over which interactions are performed on which parts of a page, thereby affecting derived input alphabets: Each action performed on a page corresponds to an input symbol.

- *Aggregation* – To reduce the state space of learned models, iHTML allows the aggregation of multiple user interactions executed in sequence, as long as the DOM remains unchanged in the meantime. This is particularly useful for form submissions where users have to fill in multiple input fields with data in an arbitrary order.

In this work, Mealy automata are the target automaton type, which by their definition are regular and deterministic systems for representing application behavior. For systems

with non-regular behavior, we can always learn regular approximations by stopping otherwise potentially never-ending learning processes, either by force or by using appropriate abstractions over the alphabet, thereby losing the precision of the learned models. When automatically interacting with a web application, chances are that the possible interaction space will grow over time, possibly even infinitely, which could lead to continuously growing alphabets and never-ending learning processes. iHTML provides a constraint system to exclude elements from the interaction space via user-defined conditions that are dynamically evaluated to keep the size of input alphabets finite.

Automata learning further assumes that the target systems behave deterministically. Typically, however, web applications are not necessarily expected to exhibit deterministic behavior because they often deal with dynamic data, such as randomly generated unique identifiers for database records, unpredictable user input, concurrent state modifications, and external systems whose states and responses may change over time, resulting in unpredictable outputs. In these scenarios, it is necessary to use more sophisticated techniques such as automated alphabet abstraction refinement (AAR) [44] to automatically refine the abstraction of the input alphabet when non-deterministic events occur, or to use learning algorithms that target more powerful and expressive automaton types, such as register automata [43] that can deal with data. However, these approaches have not yet been studied in the domain of web applications, and their implementation is beyond the scope of this work. Consequently, the approach presented in this work requires that instrumented web applications behave deterministically.

The approach to learning instrumented web applications uses the DOM as a sensor to observe and interpret the output of a system. Even if the application behaves deterministically, the DOM, as a projection of the application state (see Section 2.1), may still contain dynamic data that changes frequently, such as timestamps and real-time counters. For this reason, the unfiltered DOM is of limited use as a sensor for system output, since conflicting observations will cause classical automata learning algorithms to fail. iHTML itself cannot guarantee deterministic observations, but it provides a framework that allows developers to control how system output is interpreted, and it is ultimately their own responsibility to provide an appropriate abstraction over the DOM. Means to this are:

- *Projection* – Because the DOM is used for state identification, the learning setup implemented in Malwa creates an abstraction of the DOM as the system output that contains only instrumented elements. DOMs can contain information that is either irrelevant for state identification, such as developer comments, purely structural constructs for layouting, and inline JavaScript, or even parts that change randomly. Thus, iHTML lets developers control which parts of the DOM are relevant to reflect system state, and which parts of a page should be ignored, e.g., because they expose non-deterministic behavior. By further projecting observations to instrumented parts of the DOM, changes in non-instrumented parts will not lead to divergent outputs, making it easier to deal with system evolution as models remain comparable.

- *Visibility Control* – iHTML gives developers the ability to include data in the DOM for state identification purposes that is invisible to users, but visible to the learner as part of the DOM projection. For example, session data that is persisted only in client-side storage, or even data from a database that the developers deem necessary to characterize the state of the application, can be included. As a consequence, developers have greater control over how states are split in the learning process so that inferred models better reflect the semantics of the application.

Note that there exist approaches for dealing with non-deterministic behavior in the

context of automata learning, most notably the approach to AAR presented by Howar et al. in [44]. Starting with a finite abstraction over an input alphabet, once a conflicting observation is made, AAR finds the corresponding location within the query, splits the input symbol in question into two (refining the abstraction), and expands the input alphabet with the new symbol. Thus, instead of letting developers control what is and is not part of the system output via instrumentation, as proposed in this work, AAR could be used to automatically refine mined inputs when a DOM looks different than previously observed. While this method can potentially be used entirely without the need for instrumentation, there are scenarios where it would complicate matters. For example, assuming that randomly generated unique identifiers of database records are embedded in the DOM, the output alphabet could also grow infinitely, thus potentially leading to non-deterministic observations for each membership query, resulting in an equivalent number of input refinements. Although models learned in this way would be more accurate in terms of the actual system behavior, they would also grow infinitely in size, making formal verification of system properties difficult. However, an in-depth evaluation of the feasibility of AAR for learning (instrumented) web applications remains future work.

Please note that automata learning allows to infer models of web applications that are significantly more expressive than simple sitemaps, which are graph-based structures created by following links on a website, where states are identified by the URL. There are two reasons for this. On the one hand, iHTML allows a learner to interact with more elements than just with links, perform more actions on elements than just clicks, and even enter custom test data into input fields. On the other hand, sitemaps provide only a static view of the existence of pages and how they are linked, while learning captures the dynamic behavior of web applications and makes history-related dependencies between internal states visible. As a result, application states are deeply explored and inferred models represent interactions with the application from a user-level perspective, see Figure 4.7.

4.2 Benefits of Instrumentation

The use of heuristics, e.g., for finding stable XPath selectors [59, 81], is an indispensable practice in the field of black-box testing of web applications, because pages of a website can look and behave the same, even though they are structurally different, and can change quickly. In certain areas, such as crawling and data mining, the lack of control over the application's source code makes it necessary to use these heuristics to write scripts that are more likely to be invariant to system changes. Automated learning of web applications faces similar problems. However, if the source code is accessible, instrumentation can be leveraged to reliably interact with web applications without the need for heuristics. The following points provide an overview of common challenges and how instrumentation can be used to overcome them.

Finding a suitable state abstraction Identifying system states of web applications is a common challenge in process mining, and existing research also focuses on heuristics that exploit the DOM. This includes, but is not limited to, performing a similarity analysis based on the *tree edit distance* [54, 110], projecting the DOM onto a set of elements of interest [90], or comparing extracted semantic information [61]. Because they use a pure black-box approach, these techniques are automatically applicable in many scenarios and will probably be good enough in many cases, depending on the objective pursued. However, these approaches quickly reach their limits and cannot adequately identify states when applications are implemented in ways not covered by these heuristics. Learnability-

by-design and the presented instrumentation language shift the Quality Assurance (QA) measures to the source code level and allow developers to expose and thereby control which information of the DOM is relevant for state identification. As a result, the approach aligns application development with the objectives of QA engineers and allows to reliably infer behavioral models.

Finding the interaction space To build state-local alphabets, the task at hand is, given a page and its DOM, to find all interaction points, i.e., elements that can be interacted with, and provide all actions that can be performed on them. Indeed, most HTML elements have certain semantics associated to them stated in the HTML specification [107]. For example, links (`<a>`) and buttons (`<button>`) are destined to be clicked on to navigate to other pages and to trigger application-specific actions, respectively. However, developers can overwrite the default behavior using JavaScript at runtime, invalidating assumptions about the behavior of these elements. Furthermore, any other, typically non-interactable element can be made interactable and even be designed to mimic the look and feel of interactable elements. As a consequence, application states may stay undetected because a learner that only acts on the specification will not interact with other elements and models are likely to miss information relevant to quality assurance teams. Testing all possible user-level actions on all visible elements is impractical and only dramatically increases the state-local input alphabet associated to a system state. Instrumentation helps to infer behavioral models in a more targeted fashion by allowing developers to make the interaction space explicit and easy to interpret in automated testing processes.

Finding quiescent states A crucial aspect of web application testing is the determination of the application state in which user input can be processed in order to proceed with test execution. Before the advent of single page applications, this was almost trivial to determine, as fully rendered HTML documents were sent to the client with each interaction, so that waiting for quiescence was a matter of waiting for requests to finish and the page to be rendered. Web applications are becoming increasingly dynamic, and a large part of the business logic is implemented purely as client-side JavaScript. This means that DOMs can change dynamically at any time, and it requires heuristics to recognize that the application is ready to accept new input. However, heuristics always carry the risk that they may not work as expected, and thus are not applicable to every web application in every scenario. In the case of testing, this can mean that tests can become flaky, i.e., sometimes they produce the desired result and sometimes they do not. This is a particular problem in automata learning because systems are assumed to behave deterministically, meaning that non-deterministic observations will cause learning processes to fail or to not end with the expected result. Making quiescent states visible through instrumentation eliminates these problems, which guarantees a reliable deterministic test execution, see Section 4.4.5.

4.3 Learning Instrumented Web Applications

As stated in Section 4.1, web applications have the property that the possible interaction space is bounded by the DOM of the currently displayed page. This property can be exploited by learning algorithms to reduce the number of queries posed to the system, since symbols discovered during the learning process do not trigger queries for one-letter extensions of previous system states. An overview of the process is illustrated in Figure 4.1. Prerequisite for web applications to be learnable is that their HTML code is instrumented with `data-lbd-*` attributes (highlighted in red). These attributes encode

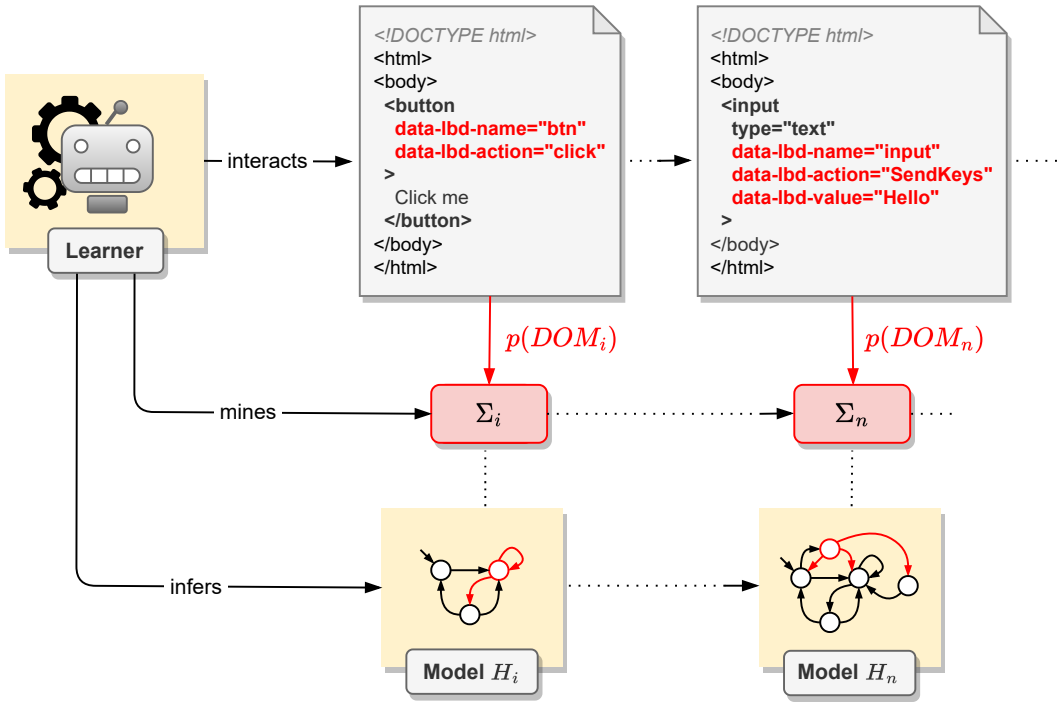


Figure 4.1: Illustration of the learnability-by-design approach.

a) input symbol definitions at an abstract level, which allows the learner to infer state-local alphabets by analyzing the DOM and

b) concrete instructions that are interpreted by the mapper to interact with the application through the browser.

By continuously interacting with an application, the DOM eventually changes and more instrumented code is uncovered, leading to more input symbols being included in the process, which results in the iterative refinement of the learned model. The learner interprets the instrumented code and navigates through the web application without any prior application-specific knowledge in an exploratory fashion.

For this approach to automated learning to work in practice, quiescent states of an application, i.e., states in which the application rests and waits for external inputs [102], must be defined and there must be an interface to query whether the system reached such a state. Because the learner is basically stepping from DOM to DOM, this is required for the learner to know when the DOM can be considered stable to perform subsequent actions. Furthermore, there needs to be an interface to reset the system to its initial state to keep membership queries independent. If these conditions are met, alphabet and automata inference can be fully automated. This approach targets learning partial Mealy machines [38], which are define as follows:

Definition 4.3.1 (Partial Mealy Machine) *Partial Mealy machines can be represented by a tuple* $M = (S, \Sigma, \Omega, s_0, \delta^*, \gamma^*)$ *where*

- S *is a finite set of states.*
- $s_0 \in S$ *is the initial state.*
- Σ *is a finite input alphabet.*

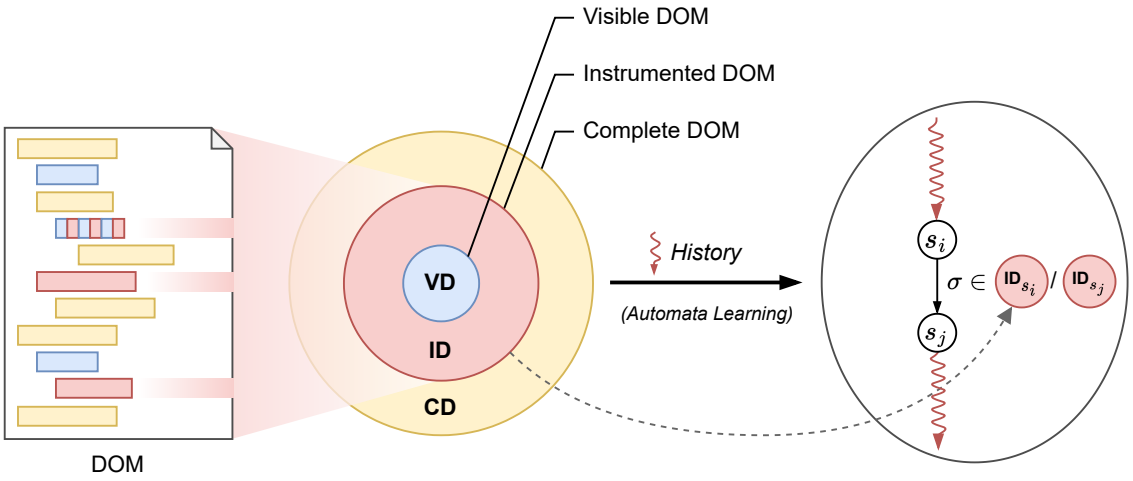


Figure 4.2: DOM-based state identification.

- Ω *is a finite output alphabet.*
- $\delta^* : S \times \Sigma_i \to S$ *is the state transition function.*
- $\gamma^* : S \times \Sigma_i \to \Omega$ *is the transition output function.*

For each state $s_i \in S$*, let* $\Sigma_i \subseteq \Sigma$ *denote the state-local alphabet of state* s_i*. Furthermore,* $\delta^*(s_i, \sigma)$ *and* $\gamma^*(s_i, \sigma)$ *are not defined for all* $\sigma \notin \Sigma_i$ *.*

4.3.1 DOM-Based Model Inference

Most web applications can architecturally be divided into three tiers: the frontend is displayed in the browser, the backend handles server-side business logic, and a data source such as a database stores application data. The state of an application can therefore be thought of as the combination of frontend state, backend state and application data. By interacting with the application through the browser, users modify the state, which in turn leads to structural changes in the DOM that affect what is displayed to the user. Consequently, analyzing the DOM allows to draw conclusions about the application state, but the accuracy depends on the amount of information within the DOM used for state identification. Figure 4.2 illustrates different projections of the DOM for a given state. The inner circle represents the *visible DOM (VD)*, which projects the DOM onto a list of elements that the user can see and interact with, such as buttons, links, and input fields. Then, the *instrumented DOM (ID)* extends the VD by explicitly marking elements that are relevant for building state-local alphabets, as well as elements that are visually hidden, but are present in the DOM. By using the iHTML, these elements are made visible to a learner by annotating them with `data-lbd-*` attributes. Finally, the *complete DOM (CD)* considers all elements in the DOM. The space of possible interactions that constitute state-local alphabets may increase with each layer, which also holds for the accuracy of learned models in terms of their ability to reflect the application state.

Considering a one-to-one relationship between a state in a model and a specific DOM, or a projection of it, one might assume that automata learning is not the appropriate approach to building state machines representing web application behavior. Indeed, this assumption is true for systems where states are independent of previous actions, such as navigation graphs spanned by links of a static website. In these cases, one can build a

labeled transition system where two states are considered the same if their DOMs are structurally equivalent and where actions that change the DOM represent the transitions. However, web applications are considered black-box systems from an end user perspective, and as such, no assumptions can be made about the history independence of actions. There may always exist backend or frontend logic that alters the application state in such a way that the effects are not immediately visible in the DOM, but only as the interaction progresses. For example, consider an application with role-based access management, a user $user_1$ with the role *user*, and a second user $user_2$ with the role *admin* that grants access to the admin interface. When $user_1$ logs in to the application via a login form on the home page, he or she is redirected to a personal page. However, once $user_2$ assigns the *admin* role to $user_1$, a subsequent login of $user_1$ would take him or her to the admin interface. In both situations, the DOMs containing the login form are equal, but the result of the login is a different. Such behavior cannot be represented by a state machine that uses the equivalence relation on DOMs to determine system states, because previous interactions with the system are not taken into account.

Figure 4.2 illustrates that automata learning makes these history-related dependencies, which are typically not encoded in the DOM, visible. When learning instrumented applications, the output function γ^* of the Mealy machine is defined to return a projection of the page's DOM after the input has been processed. In the figure one can see that at some place in the Mealy machine, there exists a transition (s_i, s_j) where the input is an action from the projected DOM of s_i and the output of the system, separated by a "/", is the projected DOM of the following state s_j. Because the learner can only interact with visible parts of the DOM, state-local alphabets can also only contain symbols that represent actions on visible elements. For the output however, instrumentation allows to also include invisible parts in the projected DOM, enabling developers to have more control over state splits. As a result, on the one hand, states can be associated with a non-empty set of DOMs by looking at the outputs of incoming transitions, and on the other hand, learning can lead to multiple states being associated with the same DOM. Considering the authentication example from above, the learned model would have two states that can be associated with the DOM containing the login form for $user_1$, depending on whether $user_2$ had previously been assigned the *admin* role.

Another aspect of interest is the effect of the selected DOM projection on the size of learned models. The more parts of the DOM are covered, the more states models can have, because state-local alphabets can grow in size, e.g., due to instrumentation, and because even small structural DOM differences can lead to state splits, especially when considering CD. Also, the likelihood of non-deterministic observations may increase if more parts of the DOM are included for the output during learning, since parts of it may change randomly, such as product recommendations in a web shop. Note that using iHTML does not prevent non-deterministic observations, as the language itself only provides a framework for code instrumentation, and developers are still responsible for not annotating parts of the DOM that are determined to show such behavior at runtime.

4.3.2 Mining System Inputs

To infer state-local alphabets, the Minimal Adequate Teacher (MAT) model needs to be extended by an interface that allows the learner to query the system for currently enabled inputs. In the case of learning web applications through the browser, this interface already exists in the form of the DOM. There, all elements a user can interact with are present, and the task is to find an adequate abstraction that maps the DOM to a set of input symbols. Such an abstraction is given by iHTML, which annotates elements that can be

interacted with, and which also defines how they can be interacted with. Because the DOM contains both symbol definitions and information that the mapper uses to interact with the system, there exists a projection function p that maps a DOM to a set of tuples $(\sigma_i, element, action, arguments)$. The tuple contains the name of an enabled input symbol σ_i, the associated *element* in the DOM to interact with, the *action* to perform on the element and a set of *arguments* such as input values for input fields, respectively. Each time p is evaluated, the learner computes the state-local alphabet Σ_i given by the enabled inputs, and the mapper is updated accordingly.

4.3.3 Interpreting System Outputs

In the context of learning web applications, a commonly used abstraction over the output alphabet is $\Omega = \{\top, \bot\}$ to indicate whether an input could be executed successfully or not. When inferring alphabets automatically, there is no straightforward approach to determining the success of an input, as this is an application-specific interpretation and requires users to define appropriate assertion logic in the mapper implementation. Thus, in this work, the output of the system is considered to be a projection of the DOM after the system has reached a quiescent state in response to an interaction. In the projection, only annotated parts of the DOM are considered. Implementation-wise, the projection collects annotated elements in a preorder traversal of the tree spanned by the DOM. Once an annotated element is found, it is appended to a list that is eventually returned to the learner as the system output.

iHTML is designed to learn a deterministic view of the application behavior. Including unannotated parts of the DOM in the projection could lead to non-deterministic observations, e.g., when dynamic values such as the current time are displayed on the page. Consequently, this may cause consistency issues in internal data structures used by learning algorithms, or result in infinitely large automaton models. Thus, limiting the system output to annotated elements allows to keep more control over the learned behavior. This also makes the approach more robust to system evolution. Code changes that do not modify annotated parts have no impact on the projected view, meaning that state-local alphabets remain the same, thus ensuring comparability between models. Eventually, this results in the output alphabet of the learned partial Mealy machine to be

$$\Omega = D \cup \{constraint\ violation\}$$

where D is a set of projected DOMs, or rather their string representations. The symbol *constraint violation* is returned by the mapper if at a some point an element is visible and therefore is part to the state-local alphabet, but must not be interacted with in order to stay within the regular bounds of the automaton model, see Section 4.4.7.

4.3.4 Testing for Equivalence

Equivalence testing describes the practice of searching for differences between learned and actual system behavior. In the context of learning black-box systems, equivalence queries are often approximated using model-based testing methods [51]. Strategies such as the W-method [23] or the Wp-method [34], which are commonly used for conformance testing, try to prove equivalence (assuming an upper bound for additional states) via an exhaustive search that grows exponentially with the number of system states. In practice, this introduces an immense overhead when system response times are high and membership queries are expensive to execute, as it is the case for web applications. This motivates using heuristics that find counterexamples fast with a minimal amount of membership queries, which also was the main motivation for the ZULU challenge [45].

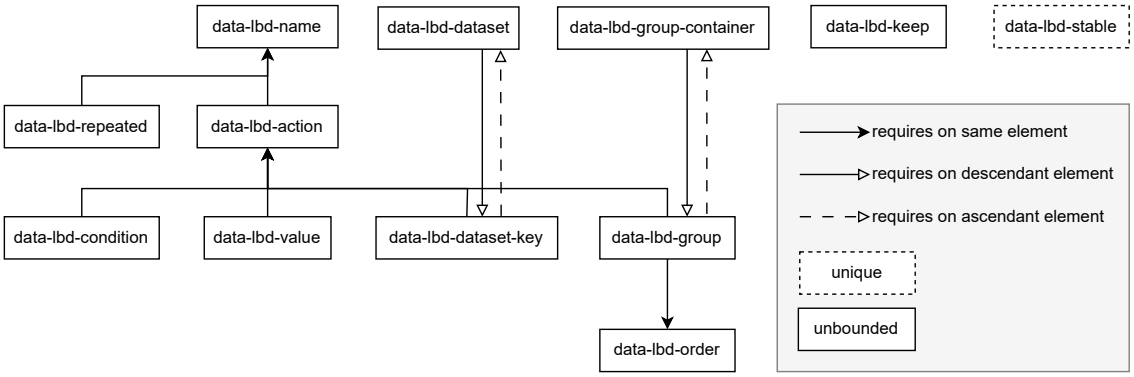


Figure 4.3: Relations of `data-lbd-*` attributes in iHTML.

In this work, a strategy is employed that combines both conformance and randomized testing to accommodate continuously growing models while keeping the number of membership queries low. The method, which exploits the properties of partial Mealy machines by posing only queries that respect state-local alphabets, works as follows:

1. Calculate a state coverage, which results in an access sequence for each state in the automaton. This guarantees that each least each state is visited once during equivalence testing.

2. For each state, execute the corresponding access sequence on the system and perform a random walk of a fixed, but user-defined length along the transitions spanned by the state-local alphabets.

3. Repeat step two for a number of m times to ensure that model boundaries are examined to a certain extend.

Let n denote the number of model states and m the number of repetitions, then a total of $n \cdot m$ membership queries is executed on the SUL during equivalence testing.

Furthermore, a heuristic is employed that takes the order of the access sequences into account. As mentioned earlier, each state in the model can be associated with a non-empty set of DOMs that are characteristic for the state by considering the outputs of incoming transitions. In practice, having multiple different DOMs represent a state is unlikely, but not impossible. This assumption is exploited by the equivalence testing strategy by sorting access sequences in descending order according to the number of different DOMs that are associated with the state the access sequence leads to in the learned model. Consequently, random walks start with those states that have a high probability of being split, which aims to reduce unnecessary test queries. This is particularly useful when using algorithms such as the TTT [48] algorithm that produce many intermediate models and are optimized for processing long counterexamples.

4.4 The iHTML Instrumentation DSL

The instrumentation of HTML code with `data` attributes is the foundation of the learnability-by-design framework. It is a common practice among JavaScript libraries and frameworks to define custom semantics for standard HTML tags in order to manipulate the behavior of the user interface. However, the intention of code instrumentation in the context of this work is not the client-side manipulation of the visible page, but to enable

third-party tools, such as Malwa, to interact with the application through a stable and standardized programmatic interface. Further, most of these libraries rely on a set of conventions, such as the naming of custom `data` attributes and the context in which they are used, and developers have to follow these conventions in order for the libraries to work as expected. Because these extensions are not part of HTML, Integrated Development Environments (IDEs) are not able to offer features such as syntax validation and autocompletion for them out-of-the-box. Therefore, the potential for misuse of these extensions and thus for incorrect or unexpected behavior of the corresponding libraries at runtime is high.

For this reason, this work introduces iHTML as a standalone DSL that extends HTML by a set of 12 custom attributes with the naming pattern `data-lbd-*` integrated into the grammar of the language. The use of iHTML, and especially the introduced attributes, guarantees that web applications are designed to be automatically learnable. The context-free grammar for iHTML is implemented in the Langium framework [57], which allows grammars to be expressed in an Extended Backus-Naur Form (EBNF) [91]. Langium generates extensions for the *Visual Studio Code* IDE [71] that enforce the syntax constraints of the language defined in the EBNF using the *language server protocol* [82]. The grammar is listed in Listing A.1 and a structural diagram that visualizes the relationships of the `data-lbd-*` attributes is illustrated in Figure 4.3. In this diagram, *unique* refers to the constraint that only a single element in the code base may have the attribute, whereas *unbounded* indicates that there are no limit restrictions. iHTML is designed with simplicity and browser compatibility in mind. It has the same syntax as HTML, which means that any iHTML document is a valid HTML document. However, this does not apply to the reverse direction, since the ordering and the position of `data-lbd-*` attributes is enforced by the grammar. Instrumentation attributes are always grouped together and placed after all other standard HTML attributes.

Because these attributes are part of the language, desired properties become *rigid* in the sense of Archimedean points in Language-Driven Engineering (LDE) [96]: IDEs enforce syntax constraints and make it impossible to create invalid code, thereby preventing misuse of language features. Moreover, since iHTML does not introduce new syntactic elements to HTML, there exists a one-to-one projection from iHTML to HTML. This eliminates the need for code generation and makes the language easier to integrate into existing projects, since switching between the two languages is a matter of switching between views within the IDE.

The grammar only takes into account the different ways in which `data-lbd-*` attributes can be combined on individual elements. However, some language properties related to the attribute usage cannot be expressed by a context-free grammar and need to be enforced by programmatic checks over the abstract syntax tree of a document instance in the IDE. This especially concerns the constraint that the values of the `data-lbd-group-container` and the `data-lbd-group` attribute must match when used in combination. In terms of Archimedean points, these properties are not rigid, but *verifiable* since the grammar is not expressive enough and they can be verified by the IDE. Arguably, alternative grammar types such as macro grammars [31], which allow parameterized non-terminals, or attribute grammars [55], which integrate language semantics directly into the grammar, might prove more suitable for expressing such constraints. The decision to use a context-free grammar was made because of its simplicity and more mature tool support. The following sections demonstrate the use of iHTML and the `data-lbd-*` attributes by examples and explain their semantics, related checks for static validation and their implications for learned models.

4.4.1 Stable Selectors

Stable selectors make it easier to maintain the testability of an application as it evolves, since elements can be located using the same selector as long as they exist in the DOM. Ideally, selectors are chosen that are still valid after the tag of an element changes, e.g. from an unordered list `<ul>` to an ordered list `<ol>`. HTML already allows elements to be annotated with a unique `id` attribute for this very purpose, e.g. to apply Cascading Style Sheets (CSS) rules to a specific element on the page.

iHTML introduces an additional attribute `data-lbd-name`, which should also contain a value that is unique across the application. The introduction of another unique property is for separation of concerns: `data-lbd-name` is used only for learning purposes to derive the name of the input symbol created for interacting with the instrumented element. This way, code that relies on the `id` attribute, e.g., to apply CSS rules or to manipulate the DOM with JavaScript, will not interfere with learning-related annotations. An example of such an annotation is the following code

```
1 <button data-lbd-name="the-button">
2   Click Me
3 </button>
```

where the `button` element is annotated with the attribute `data-lbd-name` containing the value `the-button`, which should be unique across the application. The attribute can be used to create the CSS selector

`[data-lbd-name="the-button"]`

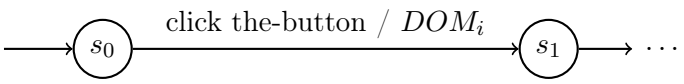
, which is short, expressive and independent from visual or other semantic properties and uniquely identifies the element on the page. Because the uniqueness constraint cannot be expressed by the grammar, there exists a check that verifies the uniqueness of used names across all files in the code base, making it also a verifiable property of the language.

4.4.2 User Interactions

There are several ways to interact with a web application through the browser, such as using a mouse and keyboard, or using the screen of the target device through touch gestures. In this work, the available interactions are limited to support actions performed with mice and keyboards. These interactions can be represented by the action set *{Hover, Click, SendKeys, Submit}*, whose values indicate that the cursor should point to an element, that an element should be clicked, that a specific text should be entered into an input field and that a form should be submitted, respectively. Elements that should be interacted with during a learning process can be annotated with the `data-lbd-action` attribute, where its value corresponds to one of the values within the action set. Once annotated, the DOM projection function p will output an abstract input symbol according to the pattern `<action> <name>` that represents the action to execute and the name of the target element. For example, by extending the button element from Section 4.4.1 in the following way

```
1 <button data-lbd-action="Click"
2         data-lbd-name="the-button">
3   Click Me
4 </button>
```

, p will output the input symbol *click the-button* and a click action on the element is associated with it within the mapper. An excerpt of an automaton with the corresponding input symbol will then have a transition



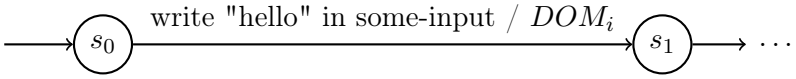
if the action can be performed on the element, resulting in the symbol being added to the state-local alphabet. The output of the system is the projected DOM of the page after the action has been executed.

4.4.3 Data Management

Most web applications offer interfaces for processing user-defined data using forms and inputs fields. For the learner to interact with these elements, test data has to be provided, which can be specified in two ways. The first alternative is by annotating corresponding input elements with the `data-lbd-value` attribute and by specifying a hard-coded string as value that is entered in the input element during learning. In the example

```
1 <input data-lbd-action="SendKeys"
2        data-lbd-value="hello"
3        data-lbd-name="some-input">
```

the string "*hello*" is automatically written into the input field with the name `some-input`. The resulting abstract input symbol is created according to the pattern `write <value> in <name>`, which means that, for this concrete example, an automaton will have the transition:



While providing raw values is a suitable approach in simple use cases, it can lead to critical test data being leaked to clients. Consequently, the framework also provides a way to specify test data that is loaded from an external source. In the following example, one can see a form used to authenticate users to the application extended with corresponding annotations to load data from an external dataset.

```
1  <form data-lbd-dataset="loginData">
2    <input type="email"
3           data-lbd-action="SendKeys"
4           data-lbd-dataset-key="email"
5           data-lbd-name="login-input-email">
6    <input type="password"
7           data-lbd-action="SendKeys"
8           data-lbd-dataset-key="password"
9           data-lbd-name="login-input-password">
10   <button data-lbd-action="Click"
11           data-lbd-name="login-button">
12     Login
13   </button>
14 </form>
```

Here, `data-lbd-dataset` specifies the name of the dataset in line 1. The approach assumes that datasets consist of a list of n-tuples, where each element within the tuple can be accessed with a named key. As such, `data-lbd-dataset-key` allows to specify the named key as a representative for the value in the tuple. For each tuple in the dataset, a new input symbol is created, allowing developers to define positive and negative examples for test inputs. Regarding the authentication example, this can be used to specify data containing

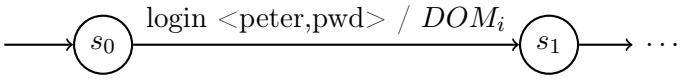
credentials for users with different roles, but also invalid credentials to learn how the system reacts to unexpected inputs. Note that in Malwa, the name of the dataset refers to a JSON file containing the data tuples that is part of the initial process configuration. During the learning process, test data is read from the JSON file and a new input symbol is created for each tuple. Each element that is annotated with `data-lbd-dataset-key` must have an element with the `data-lbd-dataset` attribute as its ancestor.

4.4.4 Grouped Actions

In the authentication example from Section 4.4.3, there are three input symbols created from the annotations. One issue that arises from this is that in the context of testing, the order of execution of the individual steps is often irrelevant. Moreover, the larger the forms become, and the more input fields there are in total, the larger the mined state-local input alphabets become. As a result, models will inevitably grow in size, learning time increases, and certain semantics may be lost or get harder to verify. To prevent a state explosion, developers can group a set of actions and define an order in which they are executed. Instead of having multiple input symbols, this ensures that only a single symbol is added to the state-local alphabet. This also allows to add more semantic context to models, which simplifies the verification of properties later. The following example extends the iHTML code of the authentication form from Section 4.4.3 with grouping attributes:

```
1  <form data-lbd-group-container="login">
2    <input type="email"
3           data-lbd-group="login"
4           data-lbd-order="1" ... />
5    <input type="password"
6           data-lbd-group="login"
7           data-lbd-order="2" ... />
8    <button data-lbd-group="login"
9            data-lbd-order="3" ... >
10     Login
11   </button>
12 </form>
```

Note that attributes from the previous example have been stripped for clarity. In the updated code, the `data-lbd-group-container` attribute has been added to the form element in line 1, which marks the parent element. Child elements are then assigned to the group using the `data-lbd-group` attribute, which is called `login` in this case. Additionally, the value of `data-lbd-order` defines the execution order within the learning process. Grouping annotations results in the creation of a single input symbol with the name of the group and the mapper executes associated actions in the specified order. In combination with the notion of datasets that is appended to the group element, a new symbol is created for each tuple according the pattern `<group> <<tuple>>`. Thus, for a given dataset `{(peter,pwd)}`, the previous example will result in a model like the following:



In terms of static validation, the following checks are applied. Elements annotated with the `data-lbd-group-container` attribute must have at least one element annotated with the `data-lbd-group` attribute in their subtree. Conversely, `group` elements cannot stand alone and must have a `group-container` element as an ancestor. In addition, the values of the attributes must match to uniquely associate elements with their containers, which

is necessary because the language allows nesting of `group-container` elements. Further, values for the `data-lbd-order` attribute may not contain duplicate values when multiple elements are in the same group container. Finally, group names must be unique throughout the codebase and distinct from any name associated with a `data-lbd-name` attribute, so that names of projected input symbols do not conflict.

4.4.5 Quiescent States

Vandraager describes the *quiescent states* of a system as the states in which the system is at rest and waits for external inputs [102]. During testing, determining when such a state is reached in order to continue with the test execution poses a challenge. Browser automation frameworks typically implement polling mechanisms to check whether certain conditions, such as the presence or absence of elements in the DOM, are met before executing further steps. The complexity of such a condition evaluation depends on the application and can increase the complexity of testing code. To make applications learnable with the proposed approach, an element node can be annotated with the `data-lbd-stable` attribute, which contains a boolean value that is updated client-side when a quiescent state is reached. It suffices that one element has this attribute, as demonstrated in the following example:

```
1 <body data-lbd-stable="true">
2   <!-- ... -->
3 </body>
```

Application developers are responsible for implementing logic to update that attribute correspondingly. The explicit notion of quiescent states

a) facilitates testers work, as condition evaluation is reduced to checking the value of the attribute, and

b) minimizes flakiness of executed queries, which reduces the chance of non-deterministic observations. In practice, the system still only waits for a predefined amount of time for the attribute to change before an action is considered to have failed.

4.4.6 Repeated Elements

Repeating similar elements, such as a list where each element has the same structure but different data, is a common pattern found in web applications. Frontend frameworks typically provide a mechanism to repeat elements based on a dataset using a template that is rendered as often as necessary. Dynamic rendering makes it difficult to add appropriate annotations that respect the unique naming constraints given by Section 4.4.1. Therefore, the instrumentation DSL allows developers to annotate these kinds of repeated elements with the `data-lbd-repeated` attribute. Considering te following list

```
1  <ul>
2    <!-- li elements are generated at runtime -->
3    <li> Apples
4      <button data-lbd-action="Click"
5              data-lbd-name="delete-button"
6              data-lbd-repeated> Delete </button>
7    </li>
8    <li> Pears
9      <button data-lbd-action="Click"
10             data-lbd-name="delete-button"
11             data-lbd-repeated> Delete </button>
```

```
12     </li>
13   </ul>
```

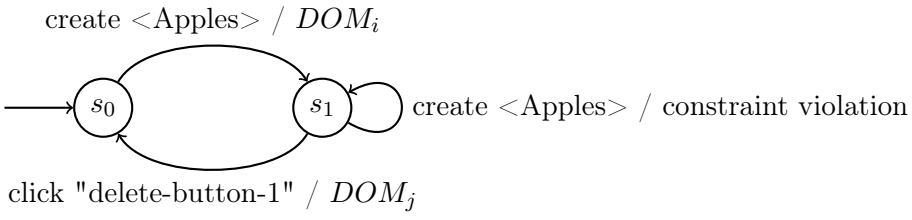
where list entries are generated at runtime based on the set `{"Apples","Pears"}`. Because these elements are repeatedly rendered, they have the same `data-lbd-name` attribute, which violates the uniqueness constraint of the annotation. In these cases, adding the `data-lbd-repeated` attribute to each element that is repeated results in the creation of a symbol for each element with respect to the index of the element in the list. The pattern for the input symbol given by p therefore is `<action> <name>-<index>`.

4.4.7 Conditional Execution

In this work, partial Mealy machines are the target automaton type for representing the behavior of web applications. However, some properties are not regular and cannot be mapped adequately to this automaton type. A basic example of this is list management, where create, read, update, and delete operations can be performed on list entries. Since lists can become potentially infinitely long, learning such a behavior results in learning processes to never terminate, because there can always be found a new state where the list is expanded by one item. Consequently, learning processes need to be constrained to learn a regular approximation of the system behavior that is sufficient enough to verify desired system properties. By adding the `data-lbd-condition` attribute to an element that has an action associated with it, the action will only be executed if the condition, provided as a JavaScript expression, evaluates to `true`. Note that the condition can only operate on the visible DOM and cannot access information collected in past states, unless it was recorded by the application itself. In the following example

```
1  <button data-lbd-condition="canExecute()"
2          data-lbd-action="Click"
3          data-lbd-name="the-button">
4    Click me
5  </button>
```

the button is only clicked if the function `canExecute()` returns `true`. Considering the list handling example, the function could check if the size of the list is below a predefined value. In case the condition evaluates to `false`, the execution is skipped and the system output is interpreted as *constraint violation* in the automaton model. This way, the observed system behavior is guaranteed to stay within the bounds of a Mealy machine. For the example from Section 4.4.6 a check could verify that the size of the list is less than two, resulting in the following automata:



As it can be seen, the execution of the symbol *create <Apples>* is prevented in s_1. However, once the existing item is deleted, see transition (s_0, s_1), the execution of the condition function returns `true` again.

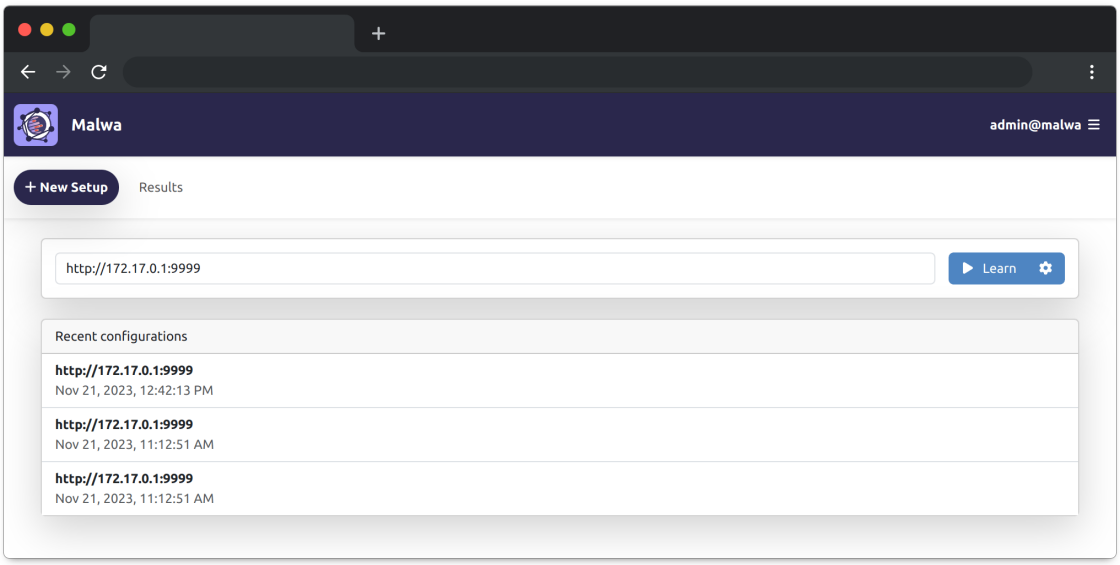


Figure 4.4: The user interface of Malwa.

4.4.8 Non-Interactive Elements

The system output function as described in Section 4.3.3 projects the DOM to a list of elements annotated with `data-lbd-*` attributes, most of which can be interacted with by the learner. However, this can lead to states in the model not being split in a desired way, e.g., when non-interactive but visible parts of the page, such as dynamic text fields or even hidden parts of the DOM, are characteristic for the identification of a state. For this purpose, the instrumentation DSL allows annotating elements with the value-less `data-lbd-keep` attribute, so that corresponding elements and their contents are respected in the output function, but are not included in state-local alphabets. Thus, in the example

```
1 <span data-lbd-keep>
2   Keep me, please.
3 </span>
```

the `<span>` element would be a part of the projected DOM and contribute to state identification. It allows developers to trigger state splitting with elements that are present in the DOM, but hidden to users.

4.5 Malwa

This work introduces the prototypical tool *Malwa* (Mostly Automated Learning of Web Applications) for learning instrumented web applications. Malwa itself is a web application with a minimal user interface illustrated in Figure 4.4 that uses LearnLib [58, 69] internally. In terms of configurations, the tool is designed to offer simplicity by default and complexity by choice, which is reflected in the following points:

- Instrumented, purely client-side applications can be learned out-of-the-box, requiring only the specification of the target URL and optionally predefined test data for forms and input fields. The system reset is performed automatically by clearing all client-side memory and reloading the specified URL.
- For applications that manage state in a database and where client state depends

on it, the tool allows to specify an additional *reset URL* that is called to perform the system reset. In this case, developers must provide such a URL as part of their public Application Programming Interface (API) where the system reset logic is implemented in code.

- The employed equivalence testing strategy is designed to adapt to the size of learned models. However, due to the black-box nature of web application testing, there may be scenarios where the generated test queries are not explorative enough to find counterexamples. For these cases, the default parameters of the equivalence testing strategy can be configured to explore hypotheses more deeply by generating more and longer test queries.

Once the *Learn* button is clicked, the target application is learned and after some time the user is presented with an inferred model similar to the one shown in Figure 4.7. For each unique projected DOM found during the learning process, a screenshot is created, which allows to associate states of a model with a set of screenshots for a more visual, interactive model analysis.

For learning automata with state-local alphabets, Malwa does not implement its own algorithm, but uses a *prefix-closure filter* [24] provided by LearnLib, which allows the use of "classic" learning algorithms targeting Mealy machines for this purpose. If at any point during a membership query a transition is not possible with respect to the state-local alphabet, the filter returns $\perp$ as the system output to indicate an invalid transition, and the rest of the query is not processed further. As a result, learned Mealy machines are complete by their original definition, and they contain a sink state in the learned model, where for every state and every input symbol that is not in the state-local alphabet, there exists a transition with $\perp$ as output label that leads to that sink state. In the following, transitions leading to the sink state are called to as $\perp$-transitions. For visualization purposes, however, only the partial Mealy machine is shown, i.e. the automaton without the sink state and without $\perp$-transitions.

4.6 Case Study: Learning TodoMVC

In the tradition of the attached publications [12] and [98], the learnability-by-design approach is demonstrated on an implementation of the TodoMVC collection [5]. As outlined in Section 1.2, the collection consists of a set of visually and behaviorally equivalent client-side single-page applications for task management, where each application is implemented using a different frontend framework. The user interface of each TodoMVC implementation is presented in Figure 4.5. Users can create tasks, mark them as completed, and remove them from the list. Additionally, the list can be filtered by the tasks' completion status, all completed tasks can be removed at once, and the completion status of all visible tasks can be toggled simultaneously.

4.6.1 Instrumentation

For demonstration purposes, the code of the React [70] implementation is extended using iHTML and learned with Malwa. An excerpt of the iHTML code is illustrated in Figure 4.6, where added `data-lbd-*` attributes are highlighted in colored boxes that correspond to user interface elements seen in Figure 4.5. Blue boxes demonstrate an example of grouped actions (see Section 4.4.4). They instruct the learner to group the actions required to create a new task by typing the string *"Apples"* into the input field and then submitting the form.

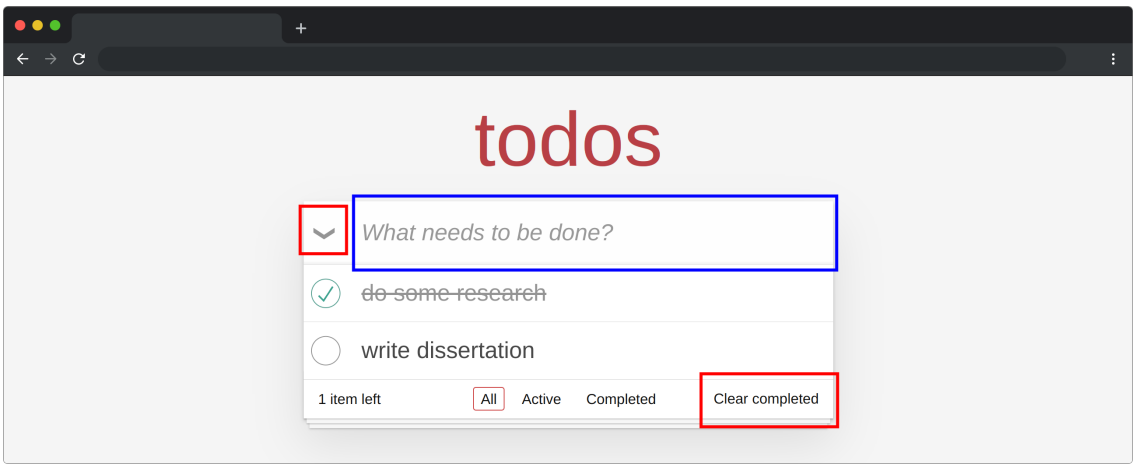


Figure 4.5: The user interface of the TodoMVC React implementation – Highlighted areas correspond to the iHTML code shown in Figure 4.6.

```
<header className="header"
        data-lbd-group-container="create">
    <h1>todos</h1>
    <form onSubmit={this.handleSubmit}
          data-lbd-name="create-form"
          data-lbd-action="Submit"
          data-lbd-group="create"
          data-lbd-order="2"
          data-lbd-condition="lbd.canCreateTodo()">
        <input
            className="new-todo"
            placeholder="What needs to be done?"
            value={this.state.newTodo}
            onChange={this.handleChange}
            autoFocus={true}
            data-lbd-name="create-input"
            data-lbd-action="SendKeys"
            data-lbd-group="create"
            data-lbd-order="1"
            data-lbd-value="Apples"
        />
    </form>
</header>
```

```
<input
    id="toggle-all"
    className="toggle-all"
    type="checkbox"
    onChange={this.toggleAll}
    checked={activeTodoCount === 0}
    data-lbd-action="Click"
    data-lbd-name="toggle-all"
/>
```

```
<button
    className="clear-completed"
    onClick={this.props.onClearCompleted}
    data-lbd-action="Click"
    data-lbd-name="clear-completed">
    Clear completed
</button>
```

Figure 4.6: Excerpt of the instrumented TodoMVC code: The task creation form (left), the switch to toggle the completion state of all tasks (top right) and the button to remove all completed tasks (bottom right).

The green box highlights the attribute for conditional form submission (see Section 4.4.7): Only if the function `lbd.canCreateTodo()` returns `true`, the actions of the group are executed. Finally, the red boxes show examples of click actions on the button to toggle the completion state of all tasks (top right) and the button to delete all completed tasks (bottom right). In order to learn the TodoMVC implementation, the following additional application-specific configurations have to be considered:

System Reset Because the system is a purely client-based application, internal state is only persisted in the browser using the `localstorage` API. Therefore, before each membership query, client-side storage is cleared and the browser is refreshed, which is the default behavior in Malwa.

Constraints TodoMVC implementations handle tasks in a mutable, potentially infinitely long list to which tasks can be added and removed. Since this is an non-regular behavior, a constraint enforces that only a fixed number n of tasks can be present at a time. The constraint has been implemented as the function `lbd.canCreateTodo` as part of the application's JavaScript code that checks the number of tasks stored in the `localstorage`. Consequently, no new tasks will be created if n tasks already exist in the list independently from their visibility.

Quiescence Following an interaction via the browser, a quiescent state of the system is considered reached as soon as the DOM has been manipulated and the browser finished rendering changes. In this example, the `data-lbd-stable` attribute on the `<body>` element is set to `false` when a user interaction is detected. Once no changes to the DOM have been observed for 100 milliseconds, the attribute is reset to `true`, indicating that the learner can take further steps. The time interval was chosen based on the experimental evaluation that user inputs have been processed, business logic has been executed, and the DOM has been updated within that timespan on the test machine.

Non-Determinism The learning process is configured to ignore the `data-reactid` attribute of all elements during the DOM projection. React, the library used to build the implementation of TodoMVC examined in this study, attaches this attribute to elements in the DOM to uniquely address them. They look similar to the following example of a list entry:

```
<li data-reactid=".0.1.2.$191000a8-47b3-47a9-a65f-8f8155927bd5">
```

Since its value is randomly generated by the library, keeping it would lead to non-deterministic observations during the learning process, so it is ignored.

4.6.2 Configuration

The experiments have been executed on a test machine which has 32 GB of RAM and a CPU with eight cores and 16 threads, each clocked between 1.9 GHz and 4.4 GHz. In addition, the TTT algorithm [48] has been selected due to the overall lower number of membership queries posed compared to related algorithms such as L* [8] or DHC [68]. All in all, the experiment consists of a series of setups. TodoMVC has been learned with the constraints that a maximum of one task ($n = 1$) and a maximum of two tasks ($n = 2$) can exist at the same time. For both scenarios, different equivalence oracles have been evaluated. This also includes the State-Local Random Walk (SLRW) equivalence testing strategy presented in Section 4.3.4, with and without access sequence ordering. The strategy has been configured to perform six random walks of length six for each access sequence in both configurations. In comparison, the W-Method and the Wp-Method have also been evaluated. The experiments have been executed against the real system for two reference runs of the learner to obtain models of TodoMVC for $n = 1$ and $n = 2$ and the SLRW oracle with the ordering heuristic (SLRW ord.) enabled. Further experiments have been executed against the already learned models loaded into memory of the test machine instead of the real system to obtain their metrics.

4.6.3 Results

Table 4.1 displays the results of the experimental evaluation, where the reference runs are highlighted in gray. One can see that for $n = 1$, a model with 13 states has been

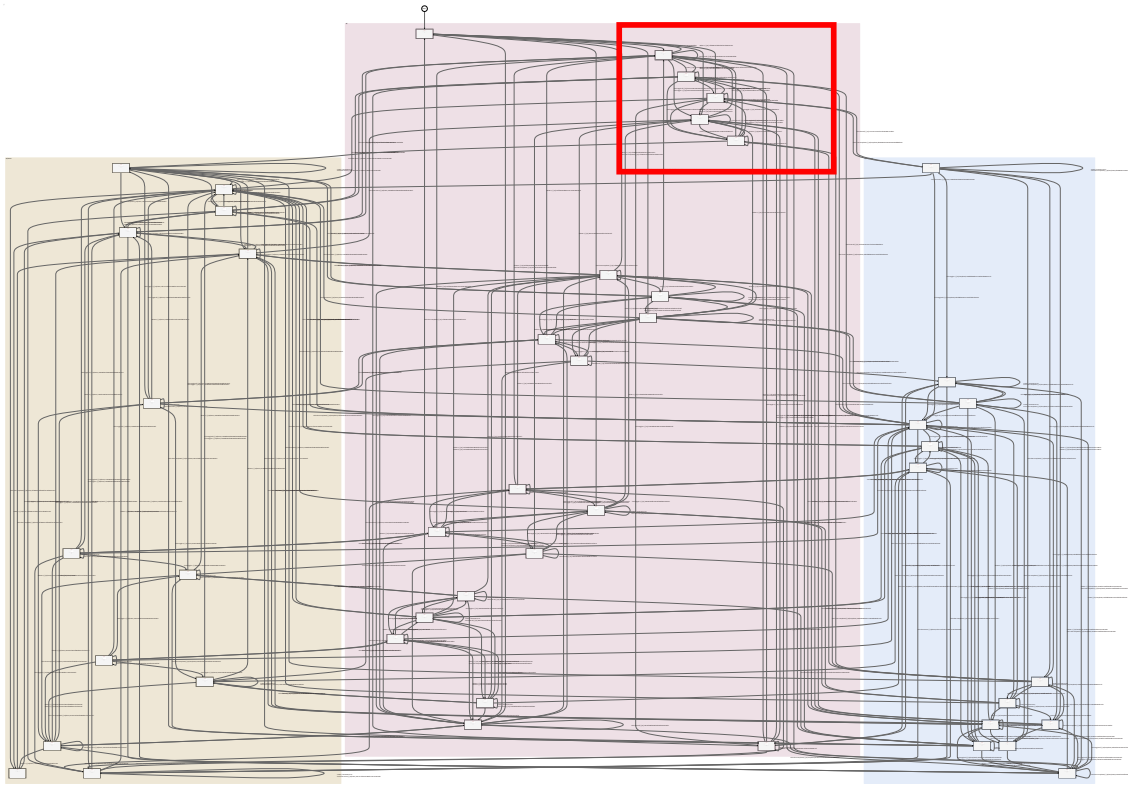


Figure 4.7: Learned model of the TodoMVC React implementation, where a maximum of two simultaneously tasks are allowed. An enlarged version of the part inside the red rectangle is shown in Figure 4.8.

learned and for $n = 2$, the processes resulted in a 48-state model. Figure 4.7 displays the 48-state model, which shows the partial Mealy machine, i.e., the automaton without the sink state and without $\perp$-transitions, representing the behavior of the TodoMVC React implementation where two tasks can be created. In Figure 4.8, a slice of the model is displayed, showing parts of the application behavior given that one of two tasks has already been created, see the state in the upper left corner.

To make the semantics of the model easier to comprehend and to simplify visual analysis, screenshots of the application associated with the corresponding DOMs taken during the learning process are displayed for each state. Because in Malwa, the system output function can be configured to also return the URL associated with a DOM, states in the model can be clustered by the path fragment of the URL. As a result, the model contains three clusters for the three list filter views and shows view-internal states as well as interactions between them: The red cluster represents the URL under which all tasks are displayed, the yellow cluster shows only active tasks, and the blue cluster shows only completed tasks. By unifying all discovered state-local alphabets, one can conclude that a total of eight input symbols in the case of $n = 1$ and ten input symbols in the case of $n = 2$ have been mined by interacting with the instrumented application. For learning TodoMVC with one task, this constitutes the input alphabet of the Mealy machine:

$$\Sigma = \{\text{create, delete-1, click toggle-1, click show-all, click show-active,}$$
$$\text{click show-completed, click toggle-all, click clear-completed}\}$$

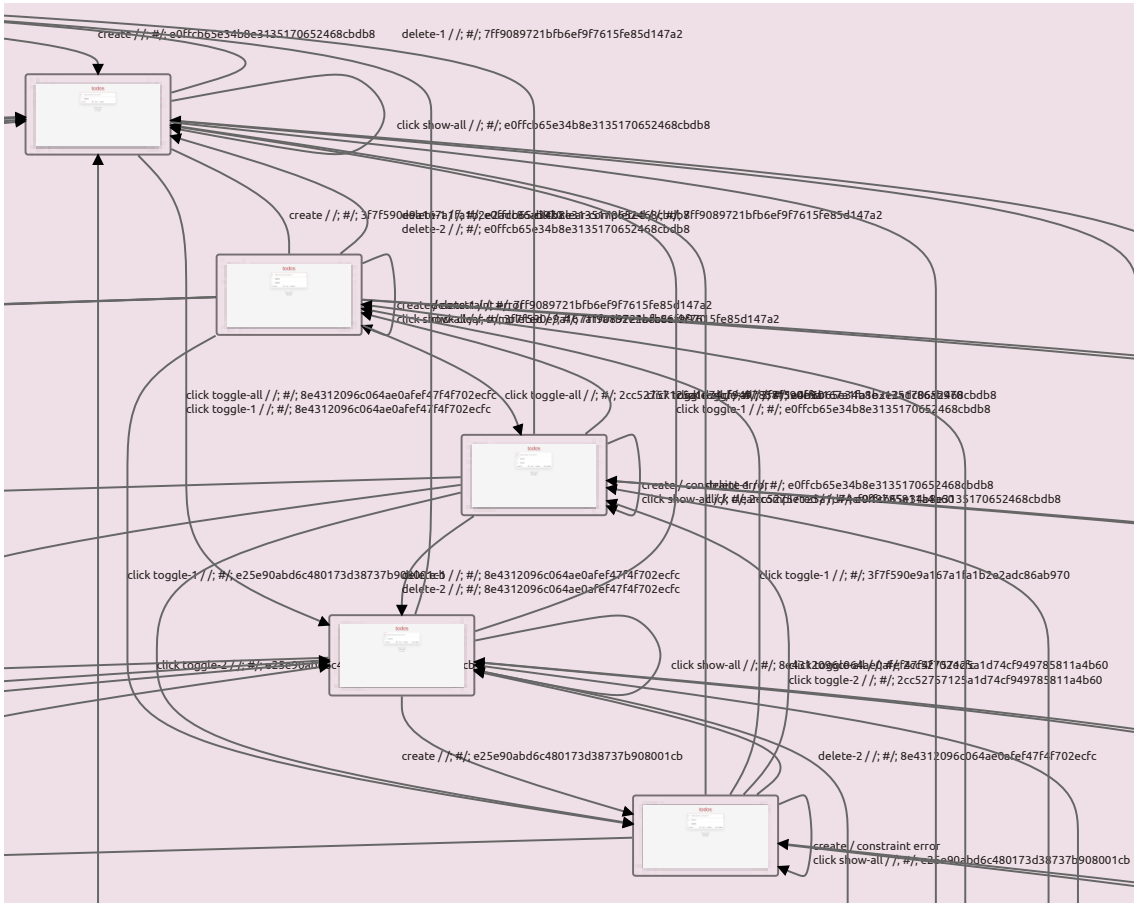


Figure 4.8: Excerpt of the learned model from Figure 4.7. The state on the upper left represents the state where one task has been created.

For $n = 2$, the alphabet is extended by {delete-2, click-toggle-2}. These cover all possible interactions with the application, where *delete-1, delete-2, click toggle-1* and *click toggle-2* refer to actions on the first two items in the task list. The output alphabet of the 13-state Mealy machine amounts to a total of 14 output symbols, 13 of which refer to the states to which only one DOM is associated, and the last one is the default output symbol *constraint violation*. Similarly, the 48-state model contains 49 output symbols where 48 refer to the states and one is the default output.

For reference, running the same learning setups with an output function that only includes the projected DOM instead of the combination of URL and DOM eventually also results in the same 13, respectively 48-state automaton. There are two reasons for this. First, the DOM alone already contains enough information to separate states because of the filter buttons at the bottom of the task list, see Figure 4.5. When a filter is active, the value of the `class` attribute of the corresponding button is set to `selected`, which is reflected in the projected DOM. At the same time, the URL is also appended with a suffix that uniquely characterizes the active filter view. Second, TodoMVC has only three static URLs, one for each filter view.

In addition, LearnLib's cache implementation answers a large number of queries in both configurations. About 43% – 45% of all queries could be answered for $n = 1$ and SLRW, and more than half of all queries, i.e., 54% – 61% for the W-Method and the Wp-Method. In comparison to this, the number of answered queries is reduced to 21% – 30% for $n = 2$

n	**EQ Oracle**	**Membership Queries**					**Ref.**	$\|\Sigma\|$	$\|\Omega\|$	**States**
		Total	*Learner*	*EQ Oracle*	*Cache*	*SUL*				
1	SLRW (ord.)	556	465	91	239	317	3	8	14	13
1	SLRW	584	490	94	263	321	3	8	14	13
1	W-Method	3257	582	2675	1986	1271	3	8	14	13
1	Wp-Method	2730	590	2140	1476	1254	3	8	14	13
2	SLRW (ord.)	2782	1971	811	829	1953	17	10	49	48
2	SLRW	3664	1873	1791	769	2895	17	10	49	48
2	W-Method	69624	2214	67410	55094	14530	21	10	49	48
2	Wp-Method	38664	2218	36446	30476	8188	21	10	49	48

Table 4.1: Statistics for learning the instrumented TodoMVC implementation.

and SLRW, and increases to 79% for both, the W-Method and the Wp-Method. These high numbers for the conformance testing methods are expected as a lot of queries are generated where there are likely a lot of redundant queries due to $\perp$-transitions.

Regarding the SLRW equivalence testing strategy, it can be concluded that it needs significantly fewer queries to find counterexamples compared to the W-Method and the Wp-Method. In this example, the SLRW (ordered) variant poses 96.6% fewer queries compared to the W-Method and 95.7% fewer queries compared the Wp-Method for $n = 1$. For $n = 2$, similar values are achieved, with 98.8% and 97.8% fewer queries compared to the W-Method and the Wp-Method, respectively. This is because, in contrast to the conformance testing methods, SLRW leverages state-locality and follows a more targeted approach to finding counterexamples, which seems to work quite well for web applications. For partial Mealy machines, W-Method and Wp-Method generate many test queries that also follow $\perp$-transitions, and thus are not as suitable for finding counterexamples quickly. Another point to mention is that the access sequence ordering heuristic reduces the amount of queries by 3.2% and 54,7% for $n = 1$ and $n = 2$, respectively. This supports the assumption that states that can be associated with multiple DOMs are more likely to split. Although the effects are not quite as visible for smaller models, the numbers indicate that the larger models get, the more impact the access sequence ordering heuristic has on the reduction of posed queries.

As a side note, learning the TodoMVC React implementation took about 26 minutes for $n = 1$ and 197 minutes for $n = 2$ on the test machine using the SLRW (ord.) equivalence testing strategy, which have been executed against the running system. As shown in Table 4.1, these are also the configurations where the least amount of membership queries have been posed to the system in both categories. In practice, the time it would take to complete the learning processes for the other configurations listed in the table is expected to correlate with the number of queries posed to the system. Especially when using the W-Method in the case of $n = 2$, the number of queries increases by an order of magnitude of 7.4, making this standard conformance testing strategy infeasible for use cases where fast feedback is desired.

Chapter 5

Learning-Based Quality Assurance in LDE

Manual instrumentation of application code can introduce a non-negligible overhead to traditional, code-centric software development. Not only can instrumentation make it more difficult to maintain large code bases, but it also shifts responsibilities from Quality Assurance (QA) engineers to frontend developers, or vice versa, by mixing test code with application code. Language-Driven Engineering (LDE) aims to align and simplify software development by allowing different stakeholders to work on separate concerns based on (graphical) Domain-Specific Languages (DSLs) using low-code and no-code tools tailored to a specific domain. However, this requires a rethinking of what QA measures are necessary, how they are integrated into the development workflow, and to what extent DSLs and code generators can be used to simplify the work of QA engineers.

Learning-based QA aims to answer the following two questions in LDE: Language engineers need to answer the question *"Does the Integrated Modeling Environment (IME) generate products correctly?"*, and application modelers want to answer the question *"Does the generated product satisfy the given functional requirements?"*. While the main concern of language engineers is to provide IMEs that reliably generate products for each imaginable model within the bounds of the language and its static semantics, application modelers focus on shipping products that adhere to given functional requirements. For both of these concerns, learnability-by-design aims to reduce the QA effort by designing languages and code generators in a way that all generated web applications on the product-level are instrumented with iHTML by default, making them automatically learnable. Because QA engineers are relieved of the manual definition of the input alphabet and the mapper implementation, all that remains to be done manually is the formulation of system properties using temporal logic to verify generated products.

Beside the instrumentation, quality control also benefits from the shift of LDE to the holistic cloud-based environment CINCO Cloud [10, 111], where meta IME as well as IMEs generated from language specifications are running in the web. By further focusing solely on web-based products, a seamless transition is achieved between all meta levels, from the initial specification of graphical DSLs to the deployment of generated products. This chapter deals with the verification of functional properties of instrumented web applications generated from graphical models using learning-based testing. Future work then discusses how iHTML can also be used to instrument web-based IMEs within CINCO Cloud to automatically verify, using learning-based testing, that language constraints are properly implemented in the modeling environment, see Section 6.4.

5.1 Controlling the Evolution

LDE is characterized by the meta-level hierarchy as described by Boßelmann in [18] which spans a tree of tools and products. At its root, a meta-modeling environment is used to generate domain-specific IMEs and each IME branches to different products. Figure 5.1 shows LDE for one particular path within the meta-level hierarchy, where so-called *path-up*

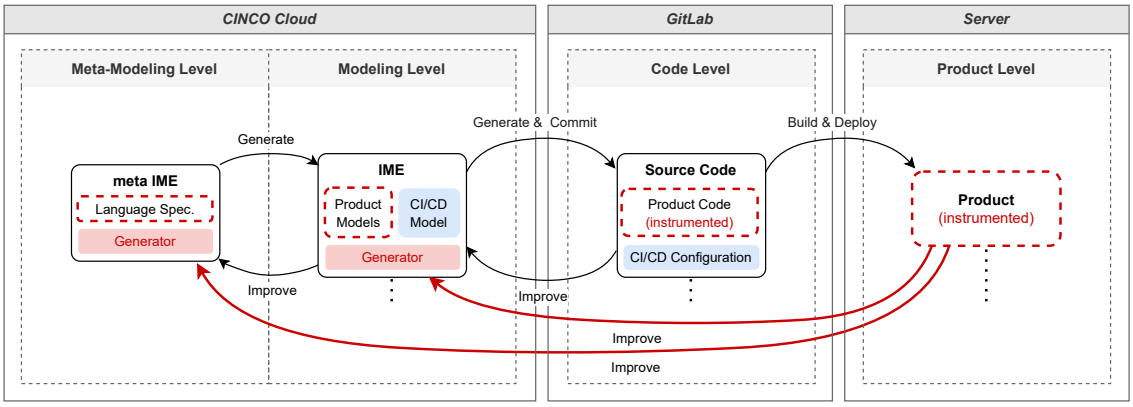


Figure 5.1: Learnability-by-Design in cloud-based LDE.

and *tree-down* effects [18] can be observed. Issues at one level, such as reported bugs, feature requests, or change requests, move up the tree in a path-up fashion to the point where the issue can be resolved. Changes propagate down the hierarchy, requiring a full re-generation of IMEs and all corresponding products. For this work, the "classic" meta-level hierarchy is extended to include a *code level* that is placed between the modeling level and the product level to accurately highlight the benefits of CINCO Cloud and to illustrate the effects of instrumentation. Tree-down effects can cause the following issues:

Meta IME → IME The meta IME allows to generate IMEs from textual language specifications and the main task is to verify if the IME corresponds to the textual specification. Generators are written manually and are therefore prone to human errors. Developers of the meta-modeling environment, i.e., the developers of CINCO Cloud, need to ensure that language constructs are correctly translated and represented in the modeling environment. Errors during this translation process allow IME users to create models that are outside of the bounds of the language specification. In this case, this can lead to issues during product generation because generators are designed to work within these bounds.

IME → Product Similar to the previous case, language engineers manually implement code generators along with the language specification at the meta-modeling level. The generator will be embedded into the target IME and it will generate the source code of the products. Again, the point of failure is the code generator, which has to translate graphical models into compileable and executable source code. Language engineers need to ensure that the generator accommodates all possible use cases. Otherwise, this could lead to application modelers to model syntactically correct models, which however translate to products that behave differently or in expected ways at runtime.

Product At the product level, QA engineers need to ensure that what application modelers modeled corresponds to given function requirements. Otherwise, end users may receive products with behavioral flaws. While model checking at the modeling level can verify system properties to some extent, it can be difficult to extract runtime properties from graphical models alone.

In this tree-down process, regression testing is an essential strategy to ensure that products still show the desired behavior after the migration. Therefore, as a part of the QA strategy in LDE,

a) code generators can be tested implicitly using black-box testing by testing generated products, which allows to verify that they have been adapted correctly to changes at the meta-modeling level, and

b) existing products need to be re-generated, re-deployed and re-learned to verify that they still adhere to given product-specific functional requirements.

In this work, the focus is latter two items of the above list by integrating learnability-by-design practices into code generators to generate instrumented web applications that can be tested automatically.

5.2 The Role of Instrumentation

The key idea of learnability-by-design in LDE is the generation of already instrumented applications, which requires the extension of DSLs and code generators to properly translate graphical models into instrumented code. By moving the responsibility for instrumenting application code from the code level to the meta-modeling level, especially as part of the code generator, instrumentation becomes a native part of the development workflow, ideally to the point where domain experts do not need to deal with it at all. Figure 5.1 illustrates where the instrumentation measures are applied and how they affect and propagate through the meta-levels:

1. Instrumentation measures can already start with the design of the language specification, which may incorporate model elements or element properties that are required for properly instrumented code, but may not have a direct impact on the behavior of the generated product.

2. Code generators at the meta-modeling level need to be designed to translate models and possibly instrumentation-specific information into valid iHTML, which requires close collaboration between language and QA engineers.

3. From the language specification, a web-based IME containing the code generator can be automatically generated and provided in CINCO Cloud. Along with the product-specific models, a model representing the Continuous Integration and Continuous Deployment (CI/CD) configuration can be specified.

4. The code generation can be triggered from within CINCO Cloud, which allows to automatically commit generated instrumented source files to a remote repository, which is, in this case, provided by GitLab.

5. With the generation, a GitLab compatible CI/CD configuration file is also generated and commited, which triggers the execution of a pipeline in which the source code is built and deployed to a remote server automatically. As a result, an instrumented application instance of the modeled product is available via the web.

6. The application instance is learned and feedback, i.e., provided by a model checker or as a result from visual model analysis can lead to refinements of product-related models, the generator on the meta-modeling level, or the language specification.

This semi-automated workflow provides an aligned approach that a) shortens feedback loops by leveraging CINCO Cloud and its built-in automation mechanisms, and b) allows both language engineers and domain experts to use the same quality control measures

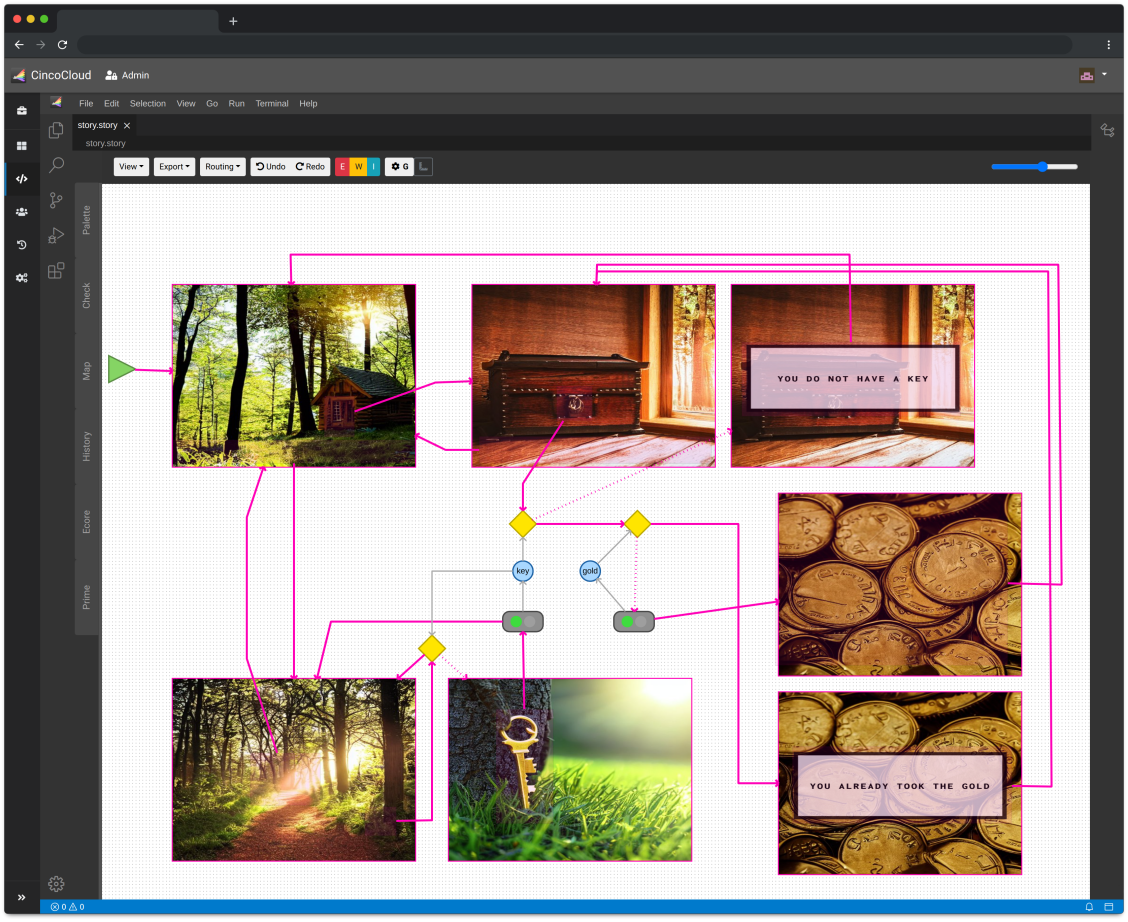


Figure 5.2: A Point-and-Click adventure using the WebStory language in CINCO Cloud. Images have been generated using Stable Diffusion [83].

by making use of instrumentation. Because instrumentation is deeply integrated into the language and the code generators, product verification can be reduced to the formulation of desired properties in temporal logic.

In particular, this allows language engineers to provide reliable IMEs to domain experts via traditional end-to-end testing, since instrumentation greatly simplifies the quality control of code generators. For that, language engineers model a range of atomic test applications which each encapsulate specific scenarios of the underlying language. Each of these applications can be generated and learned through the user interface to ensure that modeled and runtime behavior match. In case errors are detected in a generated web application, it is possible to trace what part of the code generator caused the issue while models of test applications stay small and manageable. Consequently, with enough test applications, the confidence in the correctness of the code generator increases.

5.3 Demonstration: WebStory

In Section 4.6, the code instrumentation approach has already been demonstrated on a single web application where Hypertext Markup Language (HTML) was instrumented by hand for that specific application. The manual approach requires expertise in iHTML and customization for other applications, which involves repeated manual effort. LDE can be used by incorporating aspects of iHTML into DSLs as language extensions and by

integrating them into code generators. This reduces the complexity for domain experts who graphically model web applications, and makes generated products automatically learnable: If instrumentation works for one product, it works for all.

The benefits of learnability-by-design in web-based LDE are demonstrated on *WebStory*, a graphical, CINCO-based DSL used for modeling interactive Point-and-Click adventures (in the following called *stories*), which has been used for demonstration purposes in previous publications [10, 21, 56] and which has been migrated to CINCO Cloud in [10]. In particular, the approach to instrumenting, learning, and verifying stories used in the following has already been demonstrated in one of the attached publications [21] in the context of code generation on the basis of natural language descriptions using large language models.

An exemplary story is displayed in Figure 5.2 that resembles a treasure hunt, similar to the one modeled in [56]. In this story, players start in front of a cabin in the woods and their goal is to find a golden key which is hidden in the forest. With that key, players can open a chest full of golden coins located within the cabin.

To model such adventures, WebStory comprises the following language elements: Each story is composed of *screens* that are associated with an image. On each *screen*, modelers can drag-and-drop *click areas* (pink overlays) that define clickable rectangular or circular areas within the image to indicate transitions to other *screens*. In addition, *variables* (blue circles) represent boolean values that can be set to `true` and `false` using *modify variable nodes* (gray buttons) at runtime. *Conditions* (yellow diamonds) allow to access screens conditionally based on the value of a *variable* and finally, a *start node* (green triangle) defines the initial screen. Control flow is modeled with pink edges that point from *click areas* to other *screens*, *conditions* or *modify variable nodes*. Given a graphical model, the generator generates static HTML, Cascading Style Sheets (CSS) and JavaScript files that can be opened in a web browser.

The following sections illustrate how the WebStory language and its code generator are designed so that generated stories are instrumented using iHTML to make them automatically learnable. Note that, because each DSL and code generator is designed differently, designing them to produce instrumented applications requires careful engineering which is difficult, if not impossible, to generalize.

5.3.1 Meta-Level Extensions

Generated stories only require a few language features because players can only click on specific areas of the visible images. Thus, the inferred state-local alphabets will only consist of inputs representing clicks on click areas. In order to make models easier to understand and to ease the formulation of runtime properties, the transition labels of inferred automata should express on what part of the image the player clicked. In the following, the different extensions are explained.

Language extensions The language specification for WebStory is extended to allow modelers to assign a *name* to each click area in the corresponding IME, representing on which part of the image the user clicks. The *name* property has no direct impact on the functionality of the generated stories, but it adds semantic information to the graphical models required for instrumentation. More specifically, the name is needed for the `data-lbd-name` attribute associated with click areas, which ensures that input symbols of the state-local alphabets also represent their semantics. An additional, manually implemented constraint enforces that click area names are unique per screen. Without this constraint, there could be multiple instrumented elements in the Document Object Model (DOM) with the same `data-lbd-name` attribute used to create the state-local

alphabet. In this case, only one input symbol would be created instead of one input symbol for each click area, which could result in states not being detected because the learner would only be interacting with one of the click areas.

Generator extensions The extensions of the generator focus on the mapping of click area nodes to their counterparts in the running application. Implementation-wise, click areas are generated to SVG elements placed over the currently displayed image. Considering the instrumentation, rectangular click areas are generated to

```
<rect onclick="next(...)"
      data-lbd-action="Click"
      data-lbd-name="<NAME>">
</rect>
```

indicating that these elements can be clicked on by the learner and where `<NAME>` refers to the *name* property of the corresponding click area node. The code for circular click areas is similar, using the `<ellipse>` element instead of `<rect>`.

Static code extensions The static parts of the generator have been extended by eleven lines code which handle the detection of quiescence in JavaScript. Each story is ready to accept new inputs if the image for the current screen has been loaded and is displayed. In this case, the `data-lbd-stable` attribute on the *body* element is set to *true*. When a player clicks on a click area, its value is set to *false* until the next screen is displayed.

During the learning process, the DOM is constantly projected onto a list of annotated elements to obtain the system output. The set of visible click areas alone, however, is not sufficient to accurately represent the system state. If multiple screens have the same number of click areas and the names associated with the click areas are also identical, the learner will not be able to differentiate between those screens. As a result, the learned model is likely to misrepresent the actual system behavior from a user-level perspective. Consequently, to better represent the system state using iHTML annotations, the currently displayed image is included in the projection using the `data-lbd-keep` attribute:

```
<div id="imageContainer"
     style="background-image:url(...)"
     data-lbd-keep>
</div>
```

It tells the interpreter to include the element in the DOM projection, even though users cannot interact with it directly. Since the Uniform Resource Locator (URL) of the currently displayed image is dynamically updated by the WebStory framework, the value of the `style` property will also change after each transition in the game. This way, different screens, even if they have identical click areas, produce different system outputs and the learner can better identify system states..

5.3.2 Learning WebStories

The story depicted in Figure 5.2 has been learned with Malwa in six minutes, and the inferred model can be seen in Figure 5.3. Note that although the learning algorithm used infers Mealy automata, the model is converted to a Moore automaton. On the one hand, this makes the model easier to visualize, because the DOM hashes are not displayed on the transition labels. On the other hand, it makes the model more intuitive for visual analysis because each state refers to a single DOM, which better represents user-level interactions

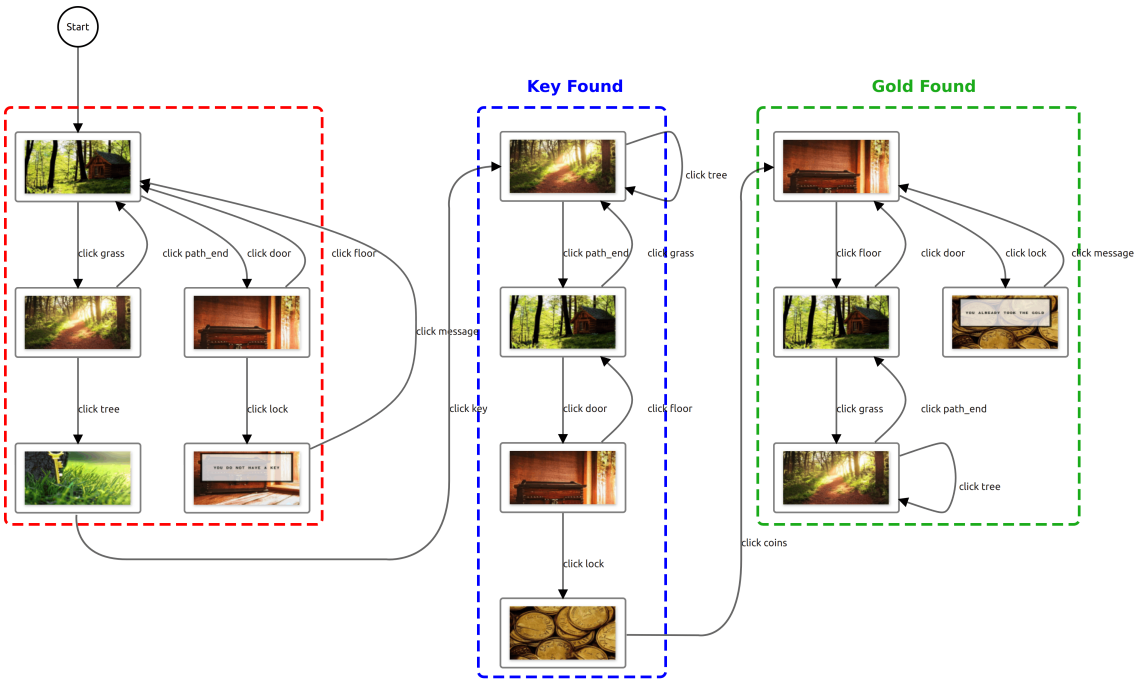


Figure 5.3: Learned runtime behavior of the WebStory shown in Figure 5.2 using Malwa. Manually layouted for compact presentation.

with the system. In contrast, Mealy machines could have states that can be associated with multiple DOMs and application screenshots because they are behaviorally equivalent. However, this could be confusing to users examining the model.

Intuitively, the model in Figure 5.3 can be interpreted as follows: Initially, users have not yet found the key, so they can navigate between the cabin and the forest, but they cannot open the chest inside the cabin (red rectangle). Once the key has been found (blue rectangle), clicking on the chest will take the user to a screen with a pile of gold. After that, users can still navigate between the cabin, the forest, and the clearing, but reopening the chest will result in the screen with the message that the gold has already been found (green rectangle). As a result, the application behavior at runtime appears to correctly represent the graphical model created in CINCO Cloud, indicating that the generator is producing the correct product for that particular model.

5.3.3 Verifying WebStories

Manual visual verification of learned models works well as long as they are sufficiently small. However, as they become large, model checking is required to automatically verify system properties. In the Moore automaton, each state can be associated with a DOM hash, which acts as its state property. To simplify the formulation of temporal logic formulas, a visual mapping can be generated that indicates which screenshot belongs to which hash, see Figure 5.4. Because QA engineers should be aware of the semantics of the individual screens, this should help them formulate temporal logic formulas that include the DOM hashes as state properties. For example, considering the learned model in Figure 5.3, the property *"gold can only be found once"* can be expressed as

```
[]((13d12f3389a322ec2c550fc02c8faf0e) ->
X([]!(13d12f3389a322ec2c550fc02c8faf0e))).
```

Figure 5.4: Legend of states from Figure 5.3 mapped to their properties.

Feedback gathered from the model checker can then be used to refine the model, the meta-level or the language itself and the established formula aids in finding regressions in subsequent iterations.

From a usability perspective, this example highlights a major problem with the current approach. State properties are not known in advance, as they are computed dynamically during the learning process by computing the hash of the DOM. This complicates the formulation of properties in temporal logic that remain stable over multiple software iterations, since even small changes in the application can make existing formulas no longer verifiable. Furthermore, by using "cryptic" looking hashes for atomic propositions, the semantics of the properties are not reflected in the formulas, making it difficult to reason with other stakeholders. Future work will address this, for example by embedding more semantic information in instrumented application code, see Section 6.2.

Chapter 6

Conclusion and Future Work

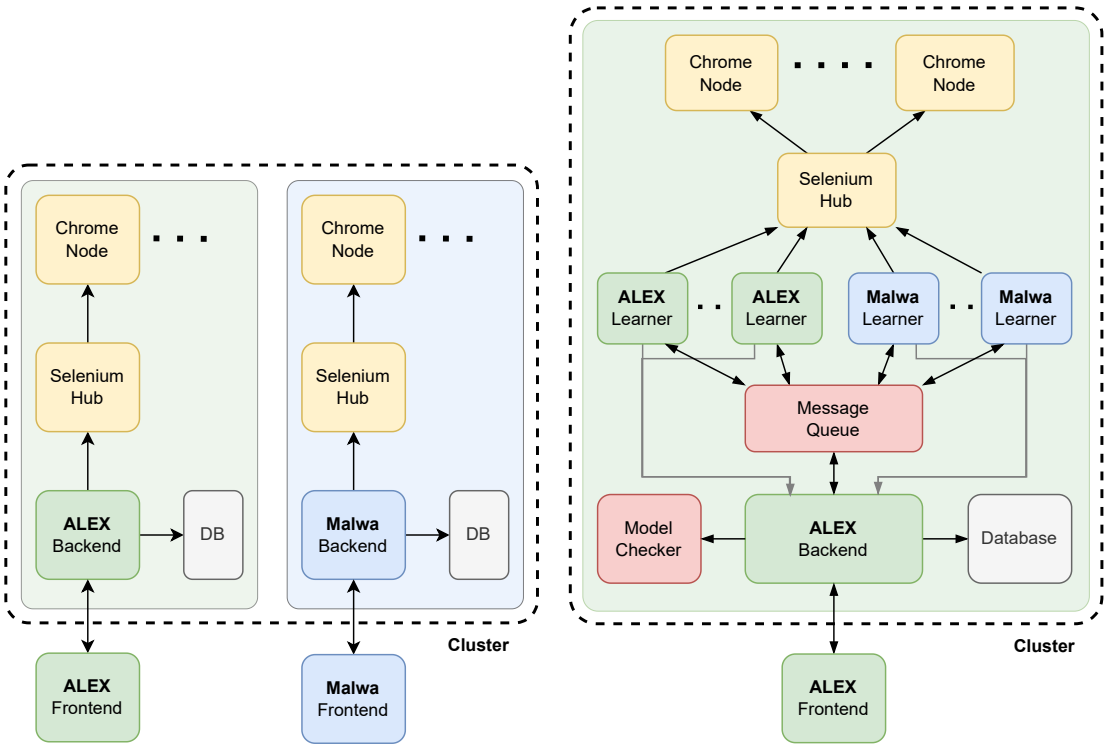
This dissertation presented the implementation of a lifelong learning framework focused on simplicity and integrability to provide a solution for the continuous quality control of web applications. It combines the results of previous research [12, 13] and highlights the synergies of the Language-Driven Engineering (LDE) ecosystem and the lifelong learning framework, building on ideas outlined in [9, 10, 98] and demonstrated in [20, 21].

First, the lifelong learning framework around the low-code tool ALEX, an all-in-one solution that lowers the entry barrier to automated learning-based testing, has been presented. From the definition of an input alphabet to the configuration of learning processes and the verification of inferred automaton models via model checking, everything takes place in a collaborative web-based environment. To take advantage of the framework's potential for web development, it has been designed to easily integrate with modern technology and development stacks by providing a set of Application Programming Interfaces (APIs) that enable the automation of lifelong learning processes in Continuous Integration and Continuous Deployments (CI/CDs) pipelines.

In addition, with *learnability-by-design*, a novel concept has been introduced that allows learning web applications via their web interface without explicit specification of an input alphabet. The key to this approach is iHTML, a textual Domain-Specific Language (DSL) that extends Hypertext Markup Language (HTML) by a set of `data-*` attributes for instrumenting web applications in a way that allows learning algorithms to incrementally mine system inputs based on information encoded in the Document Object Model (DOM) of the website. In this context, Malwa has been developed as a prototypical, web-based tool that implements a setup for learning instrumented web applications. Users only need to specify a target Uniform Resource Locator (URL) and can, if needed, configure aspects related to the learning process, such as the target browser, the parameters for the equivalence testing strategy, and the URL to call to reset the system, all of which come with sensible defaults.

Finally, this work illustrated how Quality Assurance (QA) in cloud-based LDE can be simplified by using the learnability-by-design framework. By designing DSLs and code generators for web-based products in a way that they produce instrumented HTML code, each generated application becomes instantly learnable. It has been demonstrated that this, on the one hand, enables language engineers to test their generators by learning and verifying carefully designed test applications. On the other hand, the QA effort for testing web-based products can be reduced to the verification of functional requirements through model-checking.

By working on the concepts, frameworks and tools presented in this thesis, new research topics have emerged that have the potential to further improve lifelong learning as a practice for the continuous quality control of web applications, some of which are described in the sections below. Moreover, we are aware of the technical limitations and potential challenges of using iHTML in practice, such as the implications of its use for application security, its



(a) Current state. (b) Vision.

Figure 6.1: The current system architecture of ALEX and Malwa (a) and the vision for a unified, service-oriented architecture (b).

impact on code maintainability, and its role in Dev(Sec)Ops teams for the development of continuously evolving systems. However, this dissertation focuses on demonstrating the feasibility and potential of the approach to instrumenting web applications for learnability, and thus, we refrain from discussing these aspects because they are beyond the scope of this work.

6.1 A Service-Oriented Lifelong Learning Framework

With Malwa, a tool has been implemented that allows learning instrumented web applications. From a technical perspective, however, the tool is not integrated into the existing lifelong learning framework established around ALEX. Thus, users currently do not benefit from its existing features such as the integrated model checker, measures for evolution control and its integrability with CI/CD providers. As seen in Figure 6.1a, both tools currently run completely separately in their own environments, although they share similar functionality and are architecturally alike.

Future work should focus on modularizing the framework so that the monolithic architectures of ALEX and Malwa are split and merged into an extensible landscape of microservices, as illustrated in Figure 6.1b. In this landscape

- learning processes are implemented as independent and horizontally scalable services to adapt to the current workload and user demand. There are services for learning web applications using manually implemented alphabets, as supported by ALEX (*ALEX Learner*), and services for learning instrumented web applications in the manner described in Chapter 4 (*Malwa Learner*).

- services access the same system to retrieve browser instances (*Selenium Hub*) required for executing learning processes and access shared resources from a central controller (*ALEX Backend*).

- the controller also provides the necessary public APIs that can be called in CI/CD pipelines to fully automate the phases of the lifelong learning cycle, regardless of whether the target web application is instrumented or not.

- a single web-based interface (*ALEX frontend*) that gives users the choice of either learning instrumented web applications using the minimalist configuration approach of Malwa, see Figure 4.4, or learning web applications with user-specified input alphabets, as seen in Figure 3.2.

6.2 Extensions to the Learnability-by-Design Framework

One issue with the current state of the instrumentation DSL and the corresponding DOM interpreter in Malwa is that input symbols represent atomic interactions with the user interface. As a consequence, application states are deeply explored and models tend to get large, depending on the amount of instrumented code and the complexity of the target application. In the worst case, learning processes may take too long to be feasible to execute in CI/CD pipelines, so that it is no longer worth waiting for the result because it could delay the development progress. Moreover, functional requirements are typically formulated at the level of the application's business logic, rather than at the level of atomic user interactions, such as button clicks. The latter also makes it more complicated to specify corresponding temporal logic formulas and to argue about the acceptance of certain criteria at the business logic level. To counteract these issues, iHTML would benefit from the following extensions:

Slicing would allow the learner to consider only those elements that belong to a certain *aspect*. By slicing application logic into multiple aspects, which is already a common practice when dealing with web applications, models become smaller and easier to analyze. Therefore, new attributes could be introduced that assign intractable elements one or more aspects that they belong to. Users then specify the names of aspects to be considered when configuring a learning process. As a consequence, the DOM projection function, see Section 4.3.2, which maps the DOM to a state-local alphabet, includes only elements annotated with the previously specified aspects. This way, models become smaller and faster to infer, making this approach more attractive for automated QA.

Grouping has already been introduced in Section 4.4.4 which allows grouping and ordering the execution of atomic interactions, e.g., for filling out forms in a specific order, to reduce the complexity of the learning process by shrinking the input alphabet. Currently, this functionality is limited to elements that are present in the DOM at the same time, making it impossible to combine more complex interactions that require changes to the DOM into a single input symbol. By removing this limitation, even allowing nested grouping, the abstraction level of learned models becomes a matter of configuration, which would allow to initially learn web applications on a higher behavioral abstraction level and iteratively refine models to be atomic eventually. This would result in learning hierarchical structures where states in the learned model represent entrypoints to other, more fine-granular models, comparable to the

procedural systems introduced by Frohme in [36], where multiple *procedures* are called by each other.

State Classifiers could provide more information about the application state in the DOM to ease the formulation of verifiable properties in temporal logic. Future work should investigate how more state-specific semantic information can be included in the model during the learning process. One possible approach to this is adding invisible, adequately instrumented elements to the DOM that can hold serialized state properties that can be modified at runtime, or generated from graphical models. This would allow to post-process learned models so that for each state, these properties could be extracted from the associated DOM and added to the set of state properties. Considering the WebStory example in Section 5.3, using these semantic state properties could simplify the specified formula to `[](gold -> X([]!gold))`. In contrast to using DOM hashes, this would

- simplify the creation of temporal logic formulas in advance, which would also allow to establish a fully automated learning and verification process,
- make it easier to reason about and discuss these formulas, especially in the context of acceptance testing, and
- make formulas more robust to system changes, because properties would not be affected by changes of the DOM.

However, further evaluation is needed in terms of how much of the application state can and needs to be exposed in the DOM in general, and what the implications of this are for automated learning processes.

Additionally, DSLs could be leveraged even further by transferring the idea of *Design for Verifiability* [77] to the context of LDE. The more application-specific semantics are encoded in graphical models, the easier it will be to incorporate this information into instrumented parts of generated web applications, resulting in learned models that can be verified easily. Not only that, but this also allows to generate model-checkable formulas from these models, aiming to establish a fully automated feedback loop for users in web-based LDE.

Eventually, this could even lead to a point where "traditional" active learning algorithms are no longer needed for state classification in order to build state machines of application behavior. If a projection of the application's internal state is embedded in the DOM at all times, state machines could be built on-the-fly simply by interacting with the application. As for the WebStory example in Section 5.3, all that would need to be included in the DOM is an identifier for current screen and two indicators of whether the gold and the key have been found.

6.3 Generation of Runtime Monitors

The lifelong learning framework illustrated in Chapter 3 includes the idea of generating runtime monitors from learned models that observe the System Under Learning (SUL) at runtime to detect discrepancies between learned and actual behavior. One challenge is to find a suitable abstraction for system traces that allows them to be matched to the abstract level of the input alphabet to identify the current and target state in the model. The instrumentation of HTML code, see Chapter 4, might be a step in the right direction, as abstract symbols and their mappings are not created manually, but inferred from the

website on-the-fly. As a result, there is no need to analyze system logs to find matching symbols because a monitor could leverage the same information as the learner.

For web applications, such a monitor could be implemented as a JavaScript library delivered to the browser and a server-side backend responsible for tracking user state in a given model and handling observed differences between learned and actual system behavior. On the client, the library could analyze the DOM of the website, filter instrumented elements and listen for corresponding events that the elements are annotated with. As soon as these events are triggered, the library would send the recorded action to the backend of the monitor, where the event is processed and traced on the most recent model. It still has to be evaluated if the presented learnability-by-design framework poses a suitable abstraction for this matter.

6.4 Learning-Based IME Validation

With learnability-by-design, this work has demonstrated how code generators and web applications in general can be designed to simplify learning models from the end user perspective. In LDE, it narrows the quality assurance gap between the modeling level and the product level by writing product generators that generate instrumented HTML code. However, similar challenges arise between the meta-modeling level and the modeling level that have not yet been addressed. Because Integrated Modeling Environments (IMEs) are also fully generated, they can be subject to bugs that cause modeling environments to fail to properly enforce language constraints. In this case, products may not be generated at all, or they may exhibit unforeseen behavior that is difficult to debug. Consequently, testing IMEs is a necessity to verify that language constraints are properly implemented in the modeling environment.

With the shift to CINCO Cloud, IMEs and its modeling canvases are also accessible via the web, paving the way for IMEs to become learnable as well by leveraging information encoded in the language specification. Each CINCO language contains constraints on node, container, and edge types, which elements can be nested, and which elements can be connected via drag-and-drop actions. Generating instrumented IMEs would allow a learner to incrementally explore the capabilities of the modeling canvas by creating, deleting and connecting different model elements on the canvas. Based on inferred automaton models, it would be possible to uncover whether the language constraints are correctly translated by the CINCO generator. To verify this automatically, temporal logic formulas can be generated that are compatible with the constraints specified in the language specification and the level of abstraction of the learned model. Consequently, every generated IMEs can be tested automatically without manual interference, thereby further automating and improving the quality control mechanisms in the LDE ecosystem.

List of Abbreviations

API Application Programming Interface

CI/CD Continuous Integration and Continuous Deployment

CSS Cascading Style Sheets

DAG Directed Acyclic Graph

DFA Deterministic Finite Automaton

DSL Domain-Specific Language

DOM Document Object Model

EBNF Extended Backus-Naur Form

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

IDE Integrated Development Environment

IME Integrated Modeling Environment

REST Representational State Transfer

LDE Language-Driven Engineering

LTL Linear Temporal Logic

MAT Minimal Adequate Teacher

QA Quality Assurance

SPA System of Procedural Automata

SUL System Under Learning

UML Unified Modeling Language

URL Uniform Resource Locator

References

[1] Fides Aarts. *Tomte: Bridging the Gap Between Active Learning and Real-world Systems*. IPA dissertation series. Uitgever niet vastgesteld, 2014. ISBN: 9789090284996. URL: https://books.google.de/books?id=_bH8oQEACAAJ.

[2] Fides Aarts, Paul Fiterau-Brostean, Harco Kuppens, and Frits Vaandrager. "Learning Register Automata with Fresh Value Generation". In: *Theoretical Aspects of Computing - ICTAC 2015*. Ed. by Martin Leucker, Camilo Rueda, and Frank D. Valencia. Cham: Springer International Publishing, 2015, pp. 165–183. ISBN: 978-3-319-25150-9. DOI: 10.1007/978-3-319-25150-9_11.

[3] Fides Aarts, Faranak Heidarian, Harco Kuppens, Petur Olsen, and Frits Vaandrager. "Automata Learning through Counterexample Guided Abstraction Refinement". In: *FM 2012: Formal Methods*. Ed. by Dimitra Giannakopoulou and Dominique Méry. Vol. 7436. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 10–27. ISBN: 978-3-642-32759-9. DOI: 10.1007/978-3-642-32759-9_4.

[4] Fides Aarts and Frits Vaandrager. "Learning I/O Automata". In: *CONCUR 2010 - Concurrency Theory*. Ed. by Paul Gastin and François Laroussinie. Vol. 6269. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 71–85. ISBN: 978-3-642-15375-4. DOI: 10.1007/978-3-642-15375-4_6.

[7] Silviu Andrica and George Candea. "WaRR: A tool for high-fidelity web application record and replay". In: *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*. 2011, pp. 403–410. ISBN: 978-1-4244-9232-9. DOI: 10.1109/DSN.2011.5958253.

[8] Dana Angluin. "Learning Regular Sets from Queries and Counterexamples". In: *Information and Computation* 75.2 (1987), 87—106. ISSN: 0890-5401. DOI: 10.1016/0890-5401(87)90052-6.

[9] Alexander Bainczyk, Steve Boßelmann, Marvin Krause, Marco Krumrey, Dominic Wirkner, and Bernhard Steffen. "Towards Continuous Quality Control in the Context of Language-Driven Engineering". In: *Leveraging Applications of Formal Methods, Verification and Validation. Software Engineering*. Ed. by Tiziana Margaria and Bernhard Steffen. Cham: Springer Nature Switzerland, 2022, pp. 389–406. ISBN: 978-3-031-19756-7. DOI: 10.1007/978-3-031-19756-7_22.

[10] Alexander Bainczyk, Daniel Busch, Marco Krumrey, Daniel Sami Mitwalli, Jonas Schürmann, Joel Tagoukeng Dongmo, and Bernhard Steffen. "CINCO Cloud: A Holistic Approach for Web-Based Language-Driven Engineering". In: *Leveraging Applications of Formal Methods, Verification and Validation. Software Engineering*. Ed. by Tiziana Margaria and Bernhard Steffen. Cham: Springer Nature Switzerland, 2022, pp. 407–425. ISBN: 978-3-031-19756-7. DOI: 10.1007/978-3-031-19756-7_23.

[11] Alexander Bainczyk, Alexander Schieweck, Malte Isberner, Tiziana Margaria, Johannes Neubauer, and Bernhard Steffen. "ALEX: Mixed-Mode Learning of Web Applications at Ease". In: *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications*. Ed. by Tiziana Margaria and Bernhard Steffen. Cham: Springer International Publishing, 2016, pp. 655–671. ISBN: 978-3-319-47169-3. DOI: 10.1007/978-3-319-47169-3_51.

[12] Alexander Bainczyk, Alexander Schieweck, Bernhard Steffen, and Falk Howar. "Model-Based Testing Without Models: The TodoMVC Case Study". In: *ModelEd, TestEd, TrustEd: Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday*. Ed. by Joost-Pieter Katoen, Rom Langerak, and Arend Rensink. Cham: Springer International Publishing, 2017, pp. 125–144. DOI: 10.1007/978-3-319-68270-9_7.

[13] Alexander Bainczyk, Bernhard Steffen, and Falk Howar. "Lifelong Learning of Reactive Systems in Practice". In: *The Logic of Software. A Tasting Menu of Formal Methods: Essays Dedicated to Reiner Hähnle on the Occasion of His 60th Birthday*. Ed. by Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, and Einar Broch Johnsen. Cham: Springer International Publishing, 2022, pp. 38–53. ISBN: 978-3-031-08166-8. DOI: 10.1007/978-3-031-08166-8_3.

[14] Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. "LED: Tool for Synthesizing Web Element Locators". In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2015, pp. 848–851. DOI: 10.1109/ASE.2015.110.

[15] Oliver Bauer, Johannes Neubauer, and Malte Isberner. "Model-Driven Active Automata Learning with LearnLib Studio". In: *Leveraging Applications of Formal Methods, Verification, and Validation*. Ed. by Anna-Lena Lamprecht. Cham: Springer International Publishing, 2016, pp. 128–142. ISBN: 978-3-319-51641-7. DOI: 10.1007/978-3-319-51641-7_8.

[16] Antonia Bertolino, Antonello Calabrò, Maik Merten, and Bernhard Steffen. "Never-stop Learning: Continuous Validation of Learned Models for Evolving Systems through Monitoring." In: *ERCIM News* 2012.88 (2012), pp. 28–29. URL: http://ercim-news.ercim.eu/en88/special/never-stop-learning-continuous-validation-of-learned-models-for-evolving-systems-through-monitoring.

[17] Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker, Daniel Neider, and David R. Piegdon. "libalf: The Automata Learning Framework". In: *Computer Aided Verification*. Ed. by Tayssir Touili, Byron Cook, and Paul Jackson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 360–364. ISBN: 978-3-642-14295-6. DOI: 10.1007/978-3-642-14295-6_32.

[18] Steve Boßelmann. "Evolution of Ecosystems for Language-Driven Engineering". PhD thesis. Dortmund, Germany: TU Dortmund University, 2023. DOI: 10.17877/DE290R-23218.

[19] Steve Boßelmann, Markus Frohme, Dawid Kopetzki, Michael Lybecait, Stefan Naujokat, Johannes Neubauer, Dominic Wirkner, Philip Zweihoff, and Bernhard Steffen. "DIME: A Programming-Less Modeling Environment for Web Applications". In: *Proc. of the 7th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation, Part II (ISoLA 2016)*. Vol. 9953. Lecture Notes in Computer Science. Springer, 2016, pp. 809–832. DOI: 10.1007/978-3-319-47169-3_60.

[20] Daniel Busch, Alexander Bainczyk, and Bernhard Steffen. "Towards LLM-Based System Migration in Language-Driven Engineering". In: *Engineering of Computer-Based Systems*. Ed. by Jan Kofroň, Tiziana Margaria, and Cristina Seceleanu. Cham: Springer Nature Switzerland, 2024, pp. 191–200. ISBN: 978-3-031-49252-5. DOI: 10.1007/978-3-031-49252-5_14.

[21] Daniel Busch, Gerrit Nolte, Alexander Bainczyk, and Bernhard Steffen. "ChatGPT in the Loop: A Natural Language Extension for Domain-Specific Modeling Languages". In: *Bridging the Gap Between AI and Reality*. Ed. by Bernhard Steffen. Cham: Springer Nature Switzerland, 2024, pp. 375–390. ISBN: 978-3-031-46002-9. DOI: 10.1007/978-3-031-46002-9_24.

[22] Mounir Chadli, Jin H. Kim, Kim G. Larsen, Axel Legay, Stefan Naujokat, Bernhard Steffen, and Louis-Marie Traonouez. "High-level frameworks for the specification and verification of scheduling problems". In: *Software Tools for Technology Transfer* 20.4 (2017), pp. 397–422. DOI: 10.1007/s10009-017-0466-1.

[23] T.S. Chow. "Testing Software Design Modeled by Finite-State Machines". In: *IEEE Transactions on Software Engineering* 4.3 (1978), pp. 178–187. ISSN: 1939-3520. DOI: 10.1109/TSE.1978.231496.

[24] A. Classen and P. Heymans. *Modeling with FTS: a Collection of Illustrative Examples*. Technical report Domain-Specific Optimization in Automata Learning. University of Namur, Jan. 2010. URL: https://pure.unamur.be/ws/portalfiles/portal/1051983/69416.pdf.

[26] J. Davis, K. Daniels, and R. Daniels. *Effective DevOps: Building a Culture of Collaboration, Affinity, and Tooling at Scale*. O'Reilly, 2016. ISBN: 9781491926307. URL: https://www.oreilly.com/library/view/effective-devops/9781491926291/.

[27] Vidroha Debroy, Lance Brimble, Matthew Yost, and Archana Erry. "Automating Web Application Testing from the Ground Up: Experiences and Lessons Learned in an Industrial Setting". In: *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 2018, pp. 354–362. ISBN: 978-1-5386-5012-7. DOI: 10.1109/ICST.2018.00042.

[30] Roy Thomas Fielding. "Architectural Styles and the Design of Network-based Software Architectures". PhD thesis. Irvine, California, USA: University of California, Irvine, 2000. URL: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[31] Michael J. Fischer. "Grammars with macro-like productions". In: *9th Annual Symposium on Switching and Automata Theory (swat 1968)*. 1968, pp. 131–142. DOI: 10.1109/SWAT.1968.12.

[33] Martin Fowler and Rebecca Parsons. *Domain-specific languages*. Addison-Wesley / ACM Press, 2011. ISBN: 978-0321712943. URL: http://books.google.de/books?id=ri1muolw_YwC.

[34] Yoav Freund, Michael Kearns, Dana Ron, Ronitt Rubinfeld, Robert E. Schapire, and Linda Sellie. "Efficient learning of typical finite automata from random walks". In: *STOC '93: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*. San Diego, California, United States: ACM, 1993, pp. 315–324. ISBN: 0-89791-591-7. DOI: 10.1145/167088.167191.

[35] Markus Frohme and Bernhard Steffen. "A Context-Free Symbiosis of Runtime Verification and Automata Learning". In: *Formal Methods in Outer Space: Essays Dedicated to Klaus Havelund on the Occasion of His 65th Birthday*. Ed. by Ezio Bartocci, Yliès Falcone, and Martin Leucker. Cham: Springer International Publishing, 2021, pp. 159–181. ISBN: 978-3-030-87348-6. DOI: 10.1007/978-3-030-87348-6_10.

[36] Markus Frohme and Bernhard Steffen. "Compositional Learning of Mutually Recursive Procedural Systems". In: *International Journal on Software Tools for Technology Transfer* 23.4 (2021), 521—543. ISSN: 1433-2779. DOI: 10.1007/s10009-021-00634-y.

[37] Markus Frohme and Bernhard Steffen. "Never-Stop Context-Free Learning". In: *Model Checking, Synthesis, and Learning: Essays Dedicated to Bengt Jonsson on The Occasion of His 60th Birthday*. Ed. by Ernst-Rüdiger Olderog, Bernhard Steffen, and Wang Yi. Cham: Springer International Publishing, 2021, pp. 164–185. ISBN: 978-3-030-91384-7. DOI: 10.1007/978-3-030-91384-7_9.

[38] Maren Geske. "Implementation and Performance Evaluation of an Active Learning Algorithm for Visible State-Local Alphabets". Master's thesis. TU Dortmund University, Germany, Oct. 2018.

[40] Frederik Gossen, Tiziana Margaria, Alnis Murtovi, Stefan Naujokat, and Bernhard Steffen. "DSLs for Decision Services: A Tutorial Introduction to Language-Driven Engineering". In: *Proc. of the 8th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation, Part I Modeling (ISoLA 2018)*. Vol. 11244. Lecture Notes in Computer Science. Springer, 2018, pp. 546–564. DOI: 10.1007/978-3-030-03418-4_33.

[41] Reiner Hähnle and Bernhard Steffen. "Constraint-Based Behavioral Consistency of Evolving Software Systems". In: *Machine Learning for Dynamic Software Analysis: Potentials and Limits: International Dagstuhl Seminar 16172, Dagstuhl Castle, Germany, April 24-27, 2016, Revised Papers*. Ed. by Amel Bennaceur, Reiner Hähnle, and Karl Meinke. Cham: Springer International Publishing, 2018, pp. 205–218. ISBN: 978-3-319-96562-8. DOI: 10.1007/978-3-319-96562-8_8.

[42] Mouna Hammoudi, Gregg Rothermel, and Paolo Tonella. "Why do Record/Replay Tests of Web Applications Break?" In: *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 2016, pp. 180–190. ISBN: 978-1-5090-1827-7. DOI: 10.1109/ICST.2016.16.

[43] Falk Howar, Bernhard Steffen, Bengt Jonsson, and Sofia Cassel. "Inferring Canonical Register Automata". In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Viktor Kuncak and Andrey Rybalchenko. Vol. 7148. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 251–266. ISBN: 978-3-642-27940-9. DOI: 10.1007/978-3-642-27940-9_17.

[44] Falk Howar, Bernhard Steffen, and Maik Merten. "Automata Learning with Automated Alphabet Abstraction Refinement". In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Ranjit Jhala and David Schmidt. Vol. 6538. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 263–277. ISBN: 978-3-642-18275-4. DOI: 10.1007/978-3-642-18275-4_19.

[45] Falk Howar, Bernhard Steffen, and Maik Merten. "From ZULU to RERS - Lessons Learned in the ZULU Challenge". In: *Leveraging Applications of Formal Methods, Verification, and Validation*. Ed. by Tiziana Margaria and Bernhard Steffen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 687–704. ISBN: 978-3-642-16558-0. DOI: 10.1007/978-3-642-16558-0_55.

[46] Malte Isberner, Falk Howar, and Bernhard Steffen. "Inferring Automata with State-Local Alphabet Abstractions". In: *NASA Formal Methods*. Ed. by Guillaume Brat, Neha Rungta, and Arnaud Venet. Vol. 7871. Lecture Notes in Computer Science. 2013, pp. 124–138. DOI: 10.1007/978-3-642-38088-4_9.

[47] Malte Isberner, Falk Howar, and Bernhard Steffen. "The Open-Source LearnLib". In: *Computer Aided Verification*. Ed. by Daniel Kroening and Corina S. Păsăreanu. Cham: Springer International Publishing, 2015, pp. 487–495. ISBN: 978-3-319-21690-4. DOI: 10.1007/978-3-319-21690-4_32.

[48] Malte Isberner, Falk Howar, and Bernhard Steffen. "The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning". In: *Runtime Verification*. Ed. by Borzoo Bonakdarpour and Scott A. Smolka. Cham: Springer International Publishing, 2014, pp. 307–322. ISBN: 978-3-319-11164-3. DOI: 10.1007/978-3-319-11164-3_26.

[49] Bengt Jonsson. "Learning of Automata Models Extended with Data". In: *Formal Methods for Eternal Networked Software Systems: 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011, Bertinoro, Italy, June 13-18, 2011. Advanced Lectures*. Ed. by Marco Bernardo and Valérie Issarny. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 327–349. ISBN: 978-3-642-21455-4. DOI: 10.1007/978-3-642-21455-4_10.

[50] Gijs Kant, Alfons Laarman, Jeroen Meijer, Jaco van de Pol, Stefan Blom, and Tom van Dijk. "LTSmin: High-Performance Language-Independent Model Checking". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Christel Baier and Cesare Tinelli. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 692–707. ISBN: 978-3-662-46681-0. DOI: 10.1007/978-3-662-46681-0_61.

[51] Joost-Pieter Katoen. "Labelled Transition Systems". In: *Model-Based Testing of Reactive Systems*. Ed. by Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner. Vol. 3472. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, pp. 615–616. ISBN: 978-3-540-32037-1. DOI: 10.1007/11498490_29.

[52] Michael J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory*. Cambridge, MA, USA: MIT Press, 1994. ISBN: 0-262-11193-4. DOI: 10.7551/mitpress/3897.001.0001.

[53] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley, 2008. ISBN: 978-0-470-03666-2.

[54] Yeonjung Kim, Jeahyun Park, Taehwan Kim, and Joongmin Choi. "Web Information Extraction by HTML Tree Edit Distance Matching". In: *Proceedings of the 2007 International Conference on Convergence Information Technology*. ICCIT '07. USA: IEEE Computer Society, 2007, 2455—2460. ISBN: 0769530389. DOI: 10.1109/ICCIT.2007.398.

[55] Donald E. Knuth. "Semantics of context-free languages". In: *Mathematical Systems Theory* 2.2 (June 1968), pp. 127–145. ISSN: 1433-0490. DOI: 10.1007/BF01692511.

[56] Dawid Kopetzki, Michael Lybecait, Stefan Naujokat, and Bernhard Steffen. "Towards Language-to-Language Transformation". In: *International Journal on Software Tools for Technology Transfer* (2021). ISSN: 1433-2787. DOI: 10.1007/s10009-021-00630-2.

[59] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. "Robula+: An algorithm for generating robust XPath locators for web testing". In: *Journal of Software: Evolution and Process* 28 (Mar. 2016), pp. 177–204. DOI: 10.1002/smr.1771.

[61] Jun-Wei Lin, Farn Wang, and Paul Chu. "Using Semantic Similarity in Crawling-Based Web Application Testing". In: *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 2017, pp. 138–148. DOI: 10.1109/ICST.2017.20.

[62] Wang Linzhang, Yuan Jiesong, Yu Xiaofeng, Hu Jun, Li Xuandong, and Zheng Guoliang. "Generating test cases from UML activity diagram based on Gray-box method". In: *11th Asia-Pacific Software Engineering Conference*. 2004, pp. 284–291. DOI: 10.1109/APSEC.2004.55.

[63] Alexander Maier. "Online passive learning of timed automata for cyber-physical production systems". In: *2014 12th IEEE International Conference on Industrial Informatics (INDIN)*. 2014, pp. 60–66. DOI: 10.1109/INDIN.2014.6945484.

[64] Tiziana Margaria and Bernhard Steffen. "Simplicity as a Driver for Agile Innovation". In: *Computer* 43.6 (2010), pp. 90–92. ISSN: 0018-9162. DOI: 10.1109/MC.2010.177.

[65] Miklós Maróti, Tamás Kecskés, Róbert Kereskényi, Brian Broll, Péter Völgyesi, László Jurácz, Tihamer Levendovszky, and Ákos Lédeczi. "Next generation (meta) modeling: web-and cloud-based collaborative tool infrastructure." In: *MPM@ MoDELS* 1237 (2014), pp. 41–60.

[66] Karl Meinke and Muddassar Azam Sindhu. "LBTest: A Learning-Based Testing Tool for Reactive Systems". In: *IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST), 2013*. 2013, pp. 447–454. DOI: 10.1109/ICST.2013.62.

[67] Maik Merten. "Active automata learning for real-life applications". PhD thesis. TU Dortmund University, Germany, 2013. DOI: 10.17877/DE290R-5169.

[68] Maik Merten, Falk Howar, Bernhard Steffen, and Tiziana Margaria. "Automata Learning with On-the-Fly Direct Hypothesis Construction". In: *Leveraging Applications of Formal Methods, Verification, and Validation*. Ed. by Reiner Hähnle, Jens Knoop, Tiziana Margaria, Dietmar Schreiner, and Bernhard Steffen. Communications in Computer and Information Science. Springer Berlin / Heidelberg, 2012, pp. 248–260. ISBN: 978-3-642-34780-1. DOI: 10.1007/978-3-642-34781-8_19.

[69] Maik Merten, Bernhard Steffen, Falk Howar, and Tiziana Margaria. "Next Generation LearnLib". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by K. Rustan M. Leino Parosh Aziz Abdulla. Vol. 6605. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2011, pp. 220–223. DOI: 10.1007/978-3-642-19835-9_18.

[72] Paula Montoto, Alberto Pan, Juan Raposo, Fernando Bellas, and Javier López. "Automated browsing in AJAX websites". In: *Data & Knowledge Engineering* 70.3 (2011), pp. 269–283. ISSN: 0169-023X. DOI: https://doi.org/10.1016/j.datak.2010.12.001.

[73] Stefan Naujokat. "Heavy Meta. Model-Driven Domain-Specific Generation of Generative Domain-Specific Modeling Tools". PhD thesis. Dortmund, Germany: TU Dortmund University, Aug. 2017. DOI: 10.17877/DE290R-18076.

[74] Stefan Naujokat, Michael Lybecait, Dawid Kopetzki, and Bernhard Steffen. "CINCO: A Simplicity-Driven Approach to Full Generation of Domain-Specific Graphical Modeling Tools". In: *Software Tools for Technology Transfer* 20.3 (2017), pp. 327–354. DOI: 10.1007/s10009-017-0453-6.

[75] Stefan Naujokat, Louis-Marie Traonouez, Malte Isberner, Bernhard Steffen, and Axel Legay. "Domain-Specific Code Generator Modeling: A Case Study for Multi-faceted Concurrent Systems". In: *Proc. of the 6th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation, Part I (ISoLA 2014)*. Vol. 8802. Lecture Notes in Computer Science. Springer, 2014, pp. 463–480. DOI: 10.1007/978-3-662-45234-9_33.

[76] Peter Naur and Brian Randell. *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO*. 1969.

[77] Johannes Neubauer, Tiziana Margaria, and Bernhard Steffen. "Design for Verifiability: The OCS Case Study". In: *Formal Methods for Industrial Critical Systems: A Survey of Applications*. Wiley-IEEE Computer Society Press, Mar. 2013. Chap. 8, pp. 153–178. ISBN: 978-0-470-87618-3. DOI: 10.1002/9781118459898.ch8.

[78] Johannes Neubauer, Bernhard Steffen, Oliver Bauer, Stephan Windmüller, Maik Merten, Tiziana Margaria, and Falk Howar. "Automated continuous quality assurance". In: *Formal Methods in Software Engineering: Rigorous and Agile Approaches (FormSERA)*. 2012, pp. 37–43. DOI: 10.1109/FormSERA.2012.6229787.

[79] Johannes Neubauer, Bernhard Steffen, and Tiziana Margaria. "Higher-Order Process Modeling: Product-Lining, Variability Modeling and Beyond". In: *Electronic Proceedings in Theoretical Computer Science* 129 (2013), pp. 259–283. DOI: 10.4204/EPTCS.129.16.

[80] Johannes Neubauer, Stephan Windmüller, and Bernhard Steffen. "Risk-Based Testing via Active Continuous Quality Control". In: *International Journal on Software Tools for Technology Transfer* 16.5 (2014), pp. 569–591. DOI: 10.1007/s10009-014-0321-6.

[81] Vu Nguyen, Thanh To, and Gia-Han Diep. "Generating and selecting resilient and maintainable locators for Web automated testing". In: *Software Testing, Verification and Reliability* 31.3 (2021). DOI: 10.1002/stvr.1760.

[84] Doron Peled, Moshe Y. Vardi, and Mihalis Yannakakis. "Black Box Checking". In: *Proc. FORTE '99*. Ed. by Jianping Wu, Samuel T. Chanson, and Qiang Gao. Boston, MA: Springer US, 1999, pp. 225–240. ISBN: 978-0-387-35578-8. DOI: 10.1007/978-0-387-35578-8_13.

[85] Christer Persson and Nur Yilmazturk. "Establishment of Automated Regression Testing at ABB: Industrial Experience Report on 'Avoiding the Pitfalls'". In: *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*. ASE '04. USA: IEEE Computer Society, 2004, 112–121. ISBN: 0769521312. DOI: 10.1109/ASE.2004.1342729.

[86] Amir Pnueli. "The temporal logic of programs". In: *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*. SFCS '77. USA: IEEE Computer Society, 1977, 46–57. DOI: 10.1109/SFCS.1977.32.

[87] Harald Raffelt, Tiziana Margaria, Bernhard Steffen, and Maik Merten. "Hybrid Test of Web Applications with Webtest". In: *TAV-WEB '08: Proceedings of the 2008 workshop on Testing, analysis, and verification of web services and applications*. Seattle, Washington: ACM, 2008, pp. 1–7. ISBN: 978-1-60558-053-1. DOI: 10.1145/1390832.1390833.

[88] Harald Raffelt, Maik Merten, Bernhard Steffen, and Tiziana Margaria. "Dynamic testing via automata learning". In: *International Journal on Software Tools for Technology Transfer (STTT)* 11.4 (2009), pp. 307–324. ISSN: 1433-2779. DOI: 10.1007/s10009-009-0120-7.

[89] Harald Raffelt, Bernhard Steffen, and Tiziana Margaria. "Dynamic Testing Via Automata Learning". In: *Hardware and Software: Verification and Testing*. Ed. by Karen Yorav. Vol. 4899. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008, pp. 136–152. DOI: 10.1007/978-3-540-77966-7_13.

[90] Matthias Schur, Andreas Roth, and Andreas Zeller. "Mining Behavior Models from Enterprise Web Applications". In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2013. Saint Petersburg, Russia: Association for Computing Machinery, 2013, 422–432. ISBN: 9781450322379. DOI: 10.1145/2491411.2491426.

[91] Roger S. Scowen. "Generic base standards". In: *Proceedings 1993 Software Engineering Standards Symposium*. 1993, pp. 25–34. DOI: 10.1109/SESS.1993.263968.

[94] Bernhard Steffen, Frederik Gossen, Stefan Naujokat, and Tiziana Margaria. "Language-Driven Engineering: From General-Purpose to Purpose-Specific Languages". In: *Computing and Software Science: State of the Art and Perspectives*. Ed. by Bernhard Steffen and Gerhard Woeginger. Vol. 10000. Lecture Notes in Computer Science. Springer, 2019. DOI: 10.1007/978-3-319-91908-9_17.

[95] Bernhard Steffen, Falk Howar, and Maik Merten. "Introduction to Active Automata Learning from a Practical Perspective". In: *Formal Methods for Eternal Networked Software Systems*. Ed. by Marco Bernardo and Valérie Issarny. Vol. 6659. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 256–296. DOI: 10.1007/978-3-642-21455-4_8.

[96] Bernhard Steffen and Stefan Naujokat. "Archimedean Points: The Essence for Mastering Change". In: *LNCS Transactions on Foundations for Mastering Change (FoMaC)* 1.1 (2016), pp. 22–46. DOI: 10.1007/978-3-319-46508-1_3.

[97] Liba Svobodova. "Client/Server Model of Distributed Processing". In: *Kommunikation in Verteilten Systemen I*. Ed. by Dirk Heger, Gerhard Krüger, Otto Spaniol, and Werner Zorn. Berlin, Heidelberg: Springer Berlin Heidelberg, 1985, pp. 485–498. ISBN: 978-3-642-70285-3. DOI: 10.1007/978-3-642-70285-3_29.

[98] Tim Tegeler, Sebastian Teumert, Jonas Schürmann, Alexander Bainczyk, Daniel Busch, and Bernhard Steffen. "An Introduction to Graphical Modeling of CI/CD Workflows with Rig". In: *Leveraging Applications of Formal Methods, Verification and Validation*. Ed. by Tiziana Margaria and Bernhard Steffen. Cham: Springer International Publishing, 2021, pp. 3–17. ISBN: 978-3-030-89159-6. DOI: 10.1007/978-3-030-89159-6_1.

[99] Sebastian Teumert. "Visual Authoring of CI/CD Pipeline Configurations". Bachelor's Thesis. TU Dortmund University, Apr. 2021. URL: https://archive.org/details/visual-authoring-of-cicd-pipeline-configurations.

[102] Frits W. Vaandrager. "On the Relationship Between Process Algebra and Input/Output Automata". In: *Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science*. 1991, pp. 387–398. DOI: 10.1109/LICS.1991.151662.

[103] Vladimir Viyovic, Mirjam Maksimovic, and Branko Perisic. "Sirius: A rapid development of DSM graphical editor". In: *IEEE 18th International Conference on Intelligent Engineering Systems INES 2014*. IEEE, 2014. DOI: 10.1109/ines.2014.6909375.

[108] Stephan Windmüller, Johannes Neubauer, Bernhard Steffen, Falk Howar, and Oliver Bauer. "Active Continuous Quality Control". In: *16th International ACM SIGSOFT Symposium on Component-Based Software Engineering*. CBSE '13. New York, NY, USA: ACM SIGSOFT, 2013, pp. 111–120. DOI: 10.1145/2465449.2465469.

[109] Nils Wortmann, Malte Michel, and Stefan Naujokat. "A Fully Model-Based Approach to Software Development for Industrial Centrifuges". In: *Proc. of the 7th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation, Part II (ISoLA 2016)*. Vol. 9953. Lecture Notes in Computer Science. Springer, 2016, pp. 774–783. DOI: 10.1007/978-3-319-47169-3_58.

[110] Kaizhong Zhang and Dennis Shasha. "Simple Fast Algorithms for the Editing Distance between Trees and Related Problems". In: *SIAM Journal on Computing* 18.6 (1989), pp. 1245–1262. DOI: 10.1137/0218082.

[111] Philip Zweihoff. "Aligned and Collaborative Language-Driven Engineering". Phd thesis. Dortmund, Germany: TU Dortmund University, 2022. DOI: 10.17877/DE290R-22594.

[112] Philip Zweihoff, Stefan Naujokat, and Bernhard Steffen. "Pyro: Generating Domain-Specific Collaborative Online Modeling Environments". In: *Proc. of the 22nd Int. Conf. on Fundamental Approaches to Software Engineering (FASE 2019)*. 2019. DOI: 10.1007/978-3-030-16722-6_6.

[113] Philip Zweihoff, Tim Tegeler, Jonas Schürmann, Alexander Bainczyk, and Bernhard Steffen. "Aligned, Purpose-Driven Cooperation: The Future Way of System Development". In: *Leveraging Applications of Formal Methods, Verification and Validation*. Ed. by Tiziana Margaria and Bernhard Steffen. Cham: Springer International Publishing, 2021, pp. 426–449. ISBN: 978-3-030-89159-6. DOI: 10.1007/978-3-030-89159-6_27.

Online References

[5] Addy Osmani, Sindre Sorhus, Pascal Hartig and Stephen Sawchuk. *TodoMVC*. https://todomvc.com/. [Online; accessed 16-November-2022]. 2022.

[6] Alexander Bainczyk. *Automata Learning EXperience*. https://learnlib.github.io/alex/. [Online; accessed 16-November-2022]. 2022.

[25] Cypress.io. *JavaScript End to End Testing Framework | cypress.io testing tools*. https://www.cypress.io/. [Online; accessed 22-December-2022].

[28] Eclipse Foundation. *EMF.cloud*. https://www.eclipse.org/emfcloud/. [Accessed 18-07-2022].

[29] Eclipse Foundation. *Sirius Web*. https://www.eclipse.org/sirius/sirius-web.html. [Accessed 27-07-2022].

[32] Martin Fowler. *Language Workbenches: The Killer-App for Domain Specific Languages?* 2005. URL: http://www.martinfowler.com/articles/languageWorkbench.html.

[39] GitHub - LearnLib/automatalib: A free, open-source Java library for modeling automata, graphs, and transition systems. *https://github.com/LearnLib/automatalib*. [Online; last accessed 24-May-2023]. 2011.

[57] Langium. *Langium*. https://langium.org/. [Online; accessed 14-February-2023]. 2023.

[58] LearnLib - a framework for automata learning. *https://www.learnlib.de*. [Online; last accessed 25-June-2012]. 2011.

[60] libalf: The Automata Learning Framework. *http://libalf.informatik.rwth-aachen.de/*. [Online; last accessed 28-November-2022]. 2022.

[70] Inc. Meta Platforms. *React – A JavaScript library for building user interfaces*. https://reactjs.org/. [Online; accessed 18-January-2023]. 2023.

[71] Microsoft. *Visual Studio Code - Code Editing. Redefined*. https://code.visualstudio.com/. [Online; accessed 20-February-2023]. 2023.

[82] *Official page for Language Server Protocol*. https://microsoft.github.io/language-server-protocol/. [Online; last accessed 12-February-2019].

[83] Stable Diffusion Online. *Stable Diffusion Online*. https://stablediffusionweb.com/. [Online; accessed 10-July-2023]. 2023.

[92] Software Freedom Conservancy (SFC). *Selenium*. https://www.selenium.dev/. [Online; accessed 22-December-2022].

[93] Software Freedom Conservancy (SFC). *Selenium IDE · Open source record and playback test automation for the web*. https://www.selenium.dev/selenium-ide/. [Online; accessed 22-December-2022].

[100] Tim Tegeler, Sebastian Teumert. *Rig | Low-Code CI/CD Modeling*. https://scce.gitlab.io/rig/. [Online; accessed 16-November-2022]. 2022.

[101] *Tomte*. https://tomte.cs.ru.nl/. [Online; last accessed 28-November-2022].

[104] W3C. *Level 1 Document Object Model Specification*. https://www.w3.org/TR/WD-DOM/cover.html. [Online; accessed 22-December-2022].

[105] W3C. *Selectors Level 3*. https://www.w3.org/TR/selectors-3/. [Online; accessed 22-December-2022].

[106] *WebGME*. https://webgme.org/. [Online; last accessed 26-July-2021].

[107] WHATWG (Apple, Google, Mozilla, Microsoft). *HTML Standard*. https://html.spec.whatwg.org/multipage/. [Online; accessed 17-February-2023]. 2023.

Appendix A

Large Figures

```
grammar IHTML

entry Model:
    document=Document;

Document:
    '<!DOCTYPE' 'html>' html=HTML;

HTML:
    '<html' attributes+=Attribute* '>' head=Head body=Body '</html>';

Head:
    '<head' attributes+=Attribute* '>' content+=Content* '</head>';

Body:
    '<body' attributes+=Attribute* lbdStable=LbdStable '>'
        content+=Content*
    '</body>';

Content:
    start=StartTag content+=Content* end=EndTag
        | empty=EmptyTag
        | text=ID
        | comment=Comment;

StartTag:
    '<' name=ID attributes+=Attribute*
        lbdAttributes=LbdAttribute? '>';

EndTag:
    '</' name=[StartTag:ID] '>';

EmptyTag:
    '<' name=ID attributes+=Attribute*
        lbdAttributes=EmptyLbdAttribute? '/>';

Attribute:
    name=ID '="' value=ID '"';

Comment:
    '<!--' text=ID '-->';

fragment BaseLbdAttribute:
    name=LbdName
        | action=LbdAction
        | value=LbdValue
        | keep=LbdKeep
        | group=LbdGroup
        | datasetKey=LbdDatasetKey;

EmptyLbdAttribute:
    BaseLbdAttribute;

LbdAttribute:
    BaseLbdAttribute
        | groupContainer=LbdGroupContainer
        | dataset=LbdDataset;

LbdName:
    'data-lbd-name="' value=ID '"' LbdRepeated?;
```

```
61
62  LbdAction:
63      'data-lbd-action="' value=('Click' | 'Hover' | 'Submit') '"'
            nameAttr=LbdName conditionAttr=LbdCondition?;
64
65  LbdValue:
66      'data-lbd-action="SendKeys"' 'data-lbd-value="' value=ID '"'
            nameAttr=LbdName conditionAttr=LbdCondition?;
67
68  LbdRepeated:
69      'data-lbd-repeated' nameAttr=LbdName;
70
71  LbdCondition:
72      'data-lbd-condition="' value=ID '"' (actionAttr=LbdAction |
            valueAttr=LbdValue | datasetKeyAttr=LbdDatasetKey);
73
74  LbdDataset:
75      'data-lbd-dataset="' value=ID '"';
76
77  LbdDatasetKey:
78      'data-lbd-action="SendKeys"' 'data-lbd-dataset-key="' value=ID '"'
             nameAttr=LbdName conditionAttr=LbdCondition?;
79
80  LbdGroupContainer:
81      'data-lbd-group-container="' value=ID '"' actionAttr=LbdAction?;
82
83  LbdGroup:
84      'data-lbd-group="' value=ID '"' orderAttr=LbdOrder (actionAttr=
            LbdAction | valueAttr=LbdValue | datasetKeyAttr=LbdDatasetKey);
85
86  LbdOrder:
87      'data-lbd-order="' value=INT '"';
88
89  LbdStable:
90      'data-lbd-stable="' value=('true' | 'false') '"';
91
92  LbdKeep returns string:
93      'data-lbd-keep';
94
95  terminal ID: /[_a-zA-Z-][\w_]*/;
96  terminal INT: /[0-9]+/;
97
98  // skip whitespace and newlines
99  hidden terminal WS: /\s+/;
100 hidden terminal ML_COMMENT: /\/\*[\s\S]*?\*\//;
101 hidden terminal SL_COMMENT: /\/\/[^\n\r]*/;
```

Listing A.1: EBNF for the iHTML language based on Langium.